Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Le concept de plugin et sa mise en oeuvre

Patris, Bruno

Award date:
2003

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. May. 2023

Facultés Universitaires Notre-Dame de la Paix
Namur
Institut d'Informatique

Le concept de *plugin* et sa mise en oeuvre

par Bruno Patris

Mémoire présenté en vue
de l'obtention du grade
de Maître en Informatique

Promoteur
Professeur J-P. Leclercq

Année académique 2002-2003

LBS 10027835

Résumé

Ce mémoire tente de faire un tour des concepts que recouvre le chargement dynamique de modules dans une application. Ces modules sont communément appelés "plugins". Nous tentons dans un premier temps de cerner le contexte d'utilisation du principe du plugin en présentant trois logiciels qui le mettent en oeuvre, et une courte analyse théorique sous forme d'un modèle orienté-objet. Une analyse de ces données de départ nous conduit ensuite à l'élaboration d'une architecture générique de composants, correspondant à la structure d'un système de gestion de plugins. Nous établissons alors la correspondance entre cette architecture et les cas exposés auparavant. Nous tentons enfin d'effectuer la même démarche avec un système similaire développé sur un logiciel d'imagerie médicale, au cours d'un stage à la clinique universitaire de Mont-Godinne.

Mots clés

Plugin (Plug-in), Chargement dynamique, Architecture de composants

Abstract

This memoir attempts to propose an overview of the concepts related to the dynamic loading of modules in a software application. These modules are commonly called "plugins". We first try to give a precise idea of the context of use of the plugin principle, by presenting three softwares that implements it, and a short theoretical analysis in form of an object-oriented model. An analysis of these basic data lead us then to the development of a generic components architecture, corresponding to the structure of a plugins management system. We then bind that architecture with the cases explained before. At last we try to follow the same process with a similar system developed on a medical imaging software, during a training at the university hospital of Mont-Godinne.

Key words

Plugin (Plug-in), Dynamic loading, Components architecture

Remerciements

Je remercie Monsieur Jean-Paul Leclercq, qui a accepté d'être le promoteur de ce mémoire et d'en suivre l'état d'avancement. Merci également à son épouse pour les petites corrections de style apportées à certaines parties du document.

Je tiens particulièrement à remercier Monsieur Hubert Meurisse, Coordinateur des recherches dans le domaine de la visualisation et de l'analyse d'image à l'UCL Mont-Godinne et aux Facultés Universitaires Notre-Dame de la Paix, co-promoteur de ce mémoire. L'intérêt qu'il a porté à l'avancement de mon travail et les précieux conseils qu'il a fournis tout au long du stage furent indispensables au bon déroulement de ce dernier. Merci également pour les quelques moments de décompression qu'il nous a permis de vivre, et notamment la petite pause "Beaujolais nouveau". Merci aussi aux infirmières du service pour les petits cafés du matin bien souvent nécessaires.

Je remercie aussi Bruno Piscaglia, "Customer Services Manager" de la S.A. "Télémis", pour la grande disponibilité dont il a fait preuve en répondant à mes nombreuses questions. Ses avis m'ont été très utiles tout au long du stage.

Merci également à Karl Noben pour sa lecture attentive et critique de ce mémoire.

Je tiens tout spécialement à remercier mes amis qui se reconnaîtront et qui m'ont fait vivre cinq années d'études inoubliables. Un petit clin d'oeil particulier cependant à Pierre et Olivier pour les fameux "instants détente" dont seuls mes zygomatiques ont eu à se plaindre.

Enfin je remercie ma famille, et plus spécialement mes parents, pour leur soutien et leur sérénité durant ces cinq années. Merci à eux d'avoir toléré mes horaires extravagants et d'avoir su me donner de judicieux conseils.

Table des matières

Introduction

Depuis l'avènement de l'informatique, les performances des ordinateurs en terme de vitesse d'exécution et d'espace de stockage ne cessent de croître. Dès lors on a vu naître au cours de ces dernières années des applications de plus en plus complètes et sophistiquées, mais également de plus en plus gourmandes en ressources. On peut notamment constater que les besoins en mémoire des programmes actuels ont fortement augmenté. Par exemple, il y a dix ans, les ordinateurs étaient vendus avec 1 MB de mémoire RAM et l'application MS Word nécessitait quelque 384 kB de mémoire. Aujourd'hui, 136 MB au minimum sont nécessaires à l'exécution de MS Word XP sous Windows XP.

Une des raisons de cet accroissement constant des ressources "hardware" indispensables est le fait que l'industrie du "software" a accepté la loi de Moore du développement rapide des performances "hardware" des ordinateurs. Une autre raison à ceci réside dans ce que de nombreuses fonctionnalités intéressantes pour les utilisateurs sont constamment ajoutées aux applications. Les programmeurs sont souvent incapables d'enlever d'anciennes parties de programmes, parce que personne ne sait exactement si ces parties sont encore utilisées, ni où elles sont utilisées. Par ailleurs, la pression sous laquelle travaillent souvent les concepteurs les plonge dans une optique de résultats immédiats. A l'encontre de tous les principes enseignés dans les écoles d'ingénierie du logiciel, peu de temps est donc consacré au "design" de l'application [MMS02].

Le problème traité ici vient du fait que les morceaux de programmes correspondant à ces nouvelles fonctionnalités sont chargés à chaque exécution du programme. Le système n'est par conséquent pas utilisé de façon optimale et il peut arriver qu'il soit victime de surcharges, si d'autres applications tournent en parallèle.

Une façon de faire face à ces difficultés, prônée par Wirth en 1995 ([Wir95]) est de concevoir et maintenir des systèmes simples, qui chargent des modules seulement quand ceux-ci s'avèrent nécessaires. On aboutit alors à des architectures modulaires dans lesquelles un noyau de base de l'application communique avec des composants appelés dynamiquement et affectés à des tâches bien particulières. Des programmes fondés sur de telles architectures correspondent à ce que l'on appelle des applications basées sur des plugins (*Plug-in-Based Applications*).

Dans le cadre de mon stage aux Cliniques Universitaires de Mont-Godinne, il m'a été proposé d'adapter le viewer du logiciel d'imagerie médicale Telemis, de sorte qu'il puisse héberger des plugins. L'objectif de ce stage étant principalement opérationnel, une démarche pragmatique a été préférée au détriment de l'étude théorique du domaine.

Le principal but de ce mémoire est donc de réaliser une analyse du concept de plugin et d'en dégager les notions essentielles. Les résultats de cette étude sont regroupés sous quatre chapitres : Etat de l'art, Développements théoriques, Développements pratiques, Discussion.

Le premier chapitre, "Etat de l'art", tente de rassembler un certain nombre d'informations pertinentes sur le principe du plugin et son contexte d'utilisation. Après avoir donné une

définition du concept dont il est question, nous expliquerons les difficultés rencontrées pendant le stage et au cours des recherches préliminaires à l'analyse. Etant donné qu'il n'existe dans la littérature que très peu d'écrits théoriques sur le concept qui nous intéresse, les données regroupées prendront principalement la forme d'une présentation de cas d'école. En effet la majeure partie de ce chapitre est consacrée à l'exposé de trois logiciels. En plus d'une présentation générale du fonctionnement de ces applications, nous synthétiserons pour chacune d'entre elles les informations permettant de comprendre la façon dont y sont implémentés les systèmes de gestion des plugins. Nous clôturerons cette première partie en présentant l'essentiel d'un article proposant un modèle de conception "Orienté-Objet" pour une application basée les plugins.

Dans le second chapitre, "Développements théoriques", on trouvera dans un premier temps quelques explications très brèves sur la méthodologie employée dans l'analyse qui suivra. Une généralisation du fonctionnement classique d'une application basée sur les plugins sera alors réalisée à la lumière des informations recueillies dans l'état de l'art. Celle-ci nous permettra ensuite d'établir une architecture générique de composants pour un tel type d'application. On pourra alors valider ce modèle d'architecture en établissant la correspondance entre celui-ci et la structure des différents logiciels étudiés. Enfin, nous terminerons cette partie théorique en rassemblant des instructions-clés en Java permettant de mettre en oeuvre les composants définis auparavant.

Le troisième chapitre, "Développements pratiques", est consacré à la présentation du travail réalisé lors du stage. Nous expliquerons d'abord le contexte dans lequel est apparu le besoin d'un système de gestion de plugins, ainsi que les contraintes à respecter. Avant d'exposer les résultats concrets, nous présenterons rapidement le logiciel qui fut l'objet du travail à accomplir. Nous conclurons cette partie établissant le lien entre le travail pratique réalisé et l'analyse théorique du deuxième chapitre.

Enfin, dans la quatrième partie, intitulée "Discussion", nous tâcherons d'apporter un regard critique sur l'ensemble du travail présenté auparavant. Nous essayerons également de soulever quelques pistes pour d'éventuels travaux futurs.

Chapitre 1

Etat de l'art

1.1 Objectifs

Le but de ce premier chapitre est multiple. Premièrement il doit nous permettre d'avoir une idée claire sur ce que signifie le concept de plugin et ce que celui-ci implique. Nous préciserons donc d'abord la définition de plugin logiciel. Notons que nous mentionnerons également les difficultés que peut engendrer une étude sur ce principe. Ensuite, il est nécessaire que nous puissions nous rendre compte du contexte dans lequel il est utilisé dans les logiciels qui le mettent en oeuvre, et la façon dont il est implémenté. La présentation des cas d'école devrait pouvoir répondre à ces besoins. Enfin, il nous servira de base à une analyse qui doit nous conduire à des résultats théoriques sur plusieurs niveaux d'abstraction.

1.2 Terminologie

Commençons par remarquer que, dans la littérature, il existe deux écritures différentes, "plugin" et "plug-in". Par soucis d'unicité, nous garderons arbitrairement la première de ces deux orthographes.

Définissons dans un premier temps ce que nous entendons par "**plugin**" dans ce document. La traduction de la définition donnée par le glossaire en ligne de CNET nous donne ceci : "*Ce terme fait référence à un type de programme intégré à une application plus vaste, pour y ajouter la possibilité de réaliser une tâche particulière. L'application plus vaste en question doit être conçue de manière à accepter les plugins. Le concepteur du software édite habituellement une spécification sur le design de l'application, pour permettre à des tiers d'écrire des plugins pour cette dernière.*" [1]

Etayons cette définition en ajoutant qu'un plugin ne doit pas être connu au moment de la compilation du programme pour lequel il a été conçu. C'est pourquoi aucun code correspondant à un plugin spécifique ne se retrouve dans le code source de l'application principale. Ceci est important pour comprendre la différence entre le chargement dynamique d'un plugin et le chargement dynamique de modules (souvent appelés librairies) dans certains programmes. Alors qu'aucune trace de plugin n'est présente dans le code du logiciel principal, les sources des librairies en question font bien souvent partie du noyau de base et sont d'ailleurs explicitement nommées dans les fichiers exécutables ([MMS02]).

Une caractéristique des plugins est qu'ils doivent pouvoir être développés par des programmeurs ne connaissant rien de l'architecture du programme principal. C'est pour cette raison que la définition ci-dessus met en évidence la nécessité de l'édition de spécifications sur l'architecture de ce dernier. Celles-ci définissent les conditions indispensables pour que le plugin soit reconnu par l'application en cours d'exécution, et pour qu'il puisse communiquer de manière transparente avec les parties qui l'intéressent. On dit dans ce cas qu'elles définissent

[1] http ://www.cnet.com/Resources/Info/Glossary/Terms/plugin.html

l'(les)**interface**(s) du programme chargée(s) de la gestion des plugins.

A la lumière des explications données précédemment, nous pourrons donc conclure cette section en disant que les plugins ne sont pas des parties de l'application principale mais peuvent être vus comme de plus petites applications s'exécutant en parallèle et communiquant avec la principale.

1.3 Difficultés rencontrées

1.3.1 La contrainte "économique"

Les applications qui emploient le principe du plugin sont de plus en plus nombreuses. Parmi celles-ci, une des plus connues est sans doute le navigateur Netscape Communicator [2]. Par exemple, un plugin permettant d'y afficher des documents au format PDF a été réalisé par Adobe [3]. Parmi les autres logiciels de ce type les plus renommés citons également le lecteur multi-média Winamp [4] et l'éditeur d'images Adobe Photoshop [5]. Pour des raisons économiques évidentes il est très difficile de disposer des codes sources de telles applications. Il n'est d'ailleurs pas plus aisé de trouver des informations précises et pertinentes sur le design de tels logiciels. C'est pourquoi les applications dont nous parlerons dans ce chapitre ont une portée commerciale beaucoup moindre. Deux des trois cas étudiés font d'ailleurs partie de la catégorie de logiciels dont les sources sont publiques (Open-Source Software). Ce type de programmes représentant une part non négligeable de la production informatique actuelle, nous disposons d'une base de travail très crédible. Nous considérerons donc que les principes qui y sont implémentés sont représentatifs de la pratique des architectures sur base de plugins.

1.3.2 Disparités des types d'informations trouvées pour les études de cas

Avant de présenter les différents cas sur lesquels nous baserons l'analyse, il est important de faire remarquer que les informations disponibles sont souvent de natures très différentes.

Les articles "scientifiques" qui présentent des applications utilisant le principe du plugin exposent généralement le domaine d'application du logiciel et l'interface dont il faut se servir pour écrire des plugins. Par contre, il est rare de trouver dans ceux-ci une description de l'architecture mise en place ou du code qui lui correspond. (Voir par exemple les articles [Web] et [MCFZ99].)

En ce qui concerne les logiciels commercialisés (cfr section précédente), seuls des manuels

[2] http ://wp.netscape.com/plugins/plugins_ie.html
[3] http ://cgi.netscape.com/cgi-bin/pi_moreinfo.cgi ?PID=11623
[4] http ://www.winamp.com
[5] http ://www.adobe.com/products/photoshop/main.html

décrivant la démarche à suivre pour écrire des plugins sont disponibles. Mais ceux-ci ne sont guère intéressants pour notre analyse puisqu'ils ne nous apprennent rien sur le type de structure à adopter pour concevoir de tels programmes. Notons également qu'il est parfois difficile de se procurer des versions claires et complètes de ces derniers.

Pour commencer notre étude sur les logiciels existants, nous détenons en fait deux types de données. D'une part nous disposons du code source, et seulement du code source, de deux petites applications. D'autre part, nous nous baserons sur la description de l'architecture globale d'un logiciel de routage fonctionnant avec des plugins. Notons enfin que ces deux types d'informations sont totalement extrêmes dans la conception d'une application. Il nous sera donc indispensable d'effectuer un certain nombre d'abstractions sur celles-ci afin de pouvoir produire des structures génériques. C'est ce que nous expliquerons par la suite dans le chapitre "Développements théoriques".

1.3.3 Rareté des formalisations théoriques

Au cours de la prospection de sources théoriques développant le mécanisme du plugin, il s'est avéré que peu d'auteurs s'étaient adonnés à cet exercice. Ceci justifie d'ailleurs la nécessité d'une recherche plus théorique sur le sujet, notamment à partir d'études de cas. Une seule source convenant au but des recherches a d'ailleurs été trouvée. Celle-ci fournit un modèle de conception typique pour implémenter le principe du plugin. Ce type de modèle, appelé "*Design Pattern*" [GHJV95] est spécialement développé pour les logiciels orientés-objet. Par conséquent, il est clair que notre analyse sera fortement influencée par ce type de programmation. Remarquons déjà intuitivement que la programmation "Orientée-Objet" semble être une des plus adéquates pour mettre en oeuvre le mécanisme qui nous intéresse. En effet, celui-ci sous-entend une certaine décomposition de l'architecture logicielle. Néanmoins, l'article que nous présenterons en fin de chapitre [MMS02] propose également une brève décomposition en couches successives, qui peut également s'appliquer à d'autres types de programmation.

1.4 Présentation des trois études de cas

Introduction

Cette section est dédiée à la présentation des cas qui serviront de base à notre analyse. Ils sont au nombre de trois.

Le premier, "Nature Life's", simule les comportements et les confrontations d'animaux sur un terrain délimité. Le deuxième, "ImageJ" est un petit éditeur d'images extensible. Ces deux applications sont développées en Java. Par conséquent, il est clair que l'analyse qui suivra sera sensiblement influencée par la théorie de la programmation "Orientée-Objet".

Cependant, nous pouvons considérer que ceci constituera un avantage futur. Cela facilitera en effet la comparaison avec le système développé pendant le stage de fin d'études et expliqué dans le chapitre "Développements", lui aussi conçu sous Java. Le troisième cas exposé présente le cadre global d'un software de routage développé pour un système d'exploitation particulier. Il ne s'agit pas d'un software "Objet". Cependant, le niveau d'abstraction utilisé dans la documentation correspond très bien au niveau de la découpe idéale en composants à laquelle nous souhaitons arriver.

1.4.1 Le projet "Nature Life's"

"Nature Life's" est un simulateur de vie réalisé dans le cadre d'une Maîtrise en informatique, et totalement programmé en Java. Les concepteurs en sont Roland Bertuli et Fabrice Campanella, étudiants à l'Université de Nice lors de la réalisation du projet. Toutes les informations présentes dans cette section proviennent du site Web de l'application. [6]

Principes de la simulation

Dans cette application, des "bêtes" évoluent dans un univers virtuel et agissent en fonction de leur caractère. Les trois facteurs qui déterminent les différentes interactions dans cet univers sont l'aire de jeu, les caractéristiques des espèces et leurs comportements. Décrivons-les afin de comprendre le rôle des plugins dans l'application.

L'aire de jeu

Comme les deux autres composants principaux, l'aire de jeu est paramétrable et peut faire l'objet de certaines personnalisations. Il est ainsi possible d'y ajouter et d'en enlever des obstacles, et de configurer certaines autres propriétés du terrain. Il s'agit principalement de la gestion de propriétés météorologiques telles que le brouillard ou la neige.

Les obstacles influencent fortement les déplacements des bêtes. Certaines rebondiront dessus tandis que d'autres plus évoluées les éviteront. Ils peuvent être de deux types, rectangulaires ou ellipsoïdaux, et sont placés ou retirés à la guise de l'utilisateur grâce à l'interface de l'application.

Les propriétés intrinsèques du monde modifient fortement le comportement des différentes créatures. Il est possible de modifier l'opacité de l'air, pour cause de brouillard ou de coucher du soleil. La variation de ce facteur influencera directement la distance de vue des animaux et donc leur comportement. Par exemple, plus la distance de vue est réduite, plus les prédateurs auront du mal à trouver une proie, et plus l'esquive des obstacles sera retardée. Un autre

[6] http ://rolab.free.fr/Project/Maitrise/NatureLife

élément paramétrable est un coefficient de ralentissement des individus qui traduit le type de milieu dans lequel les bêtes se déplacent. Par exemple, dans un marais ou dans la neige, les déplacements au sol seront relativement pénibles alors que les créatures volantes n'en seront pas désavantagées. A l'inverse, il est facile d'imaginer les difficultés de déplacement de ces dernières si les conditions climatiques se dégradent (en cas de tempête notamment). De tels changements peuvent provoquer des déséquilibres considérables.

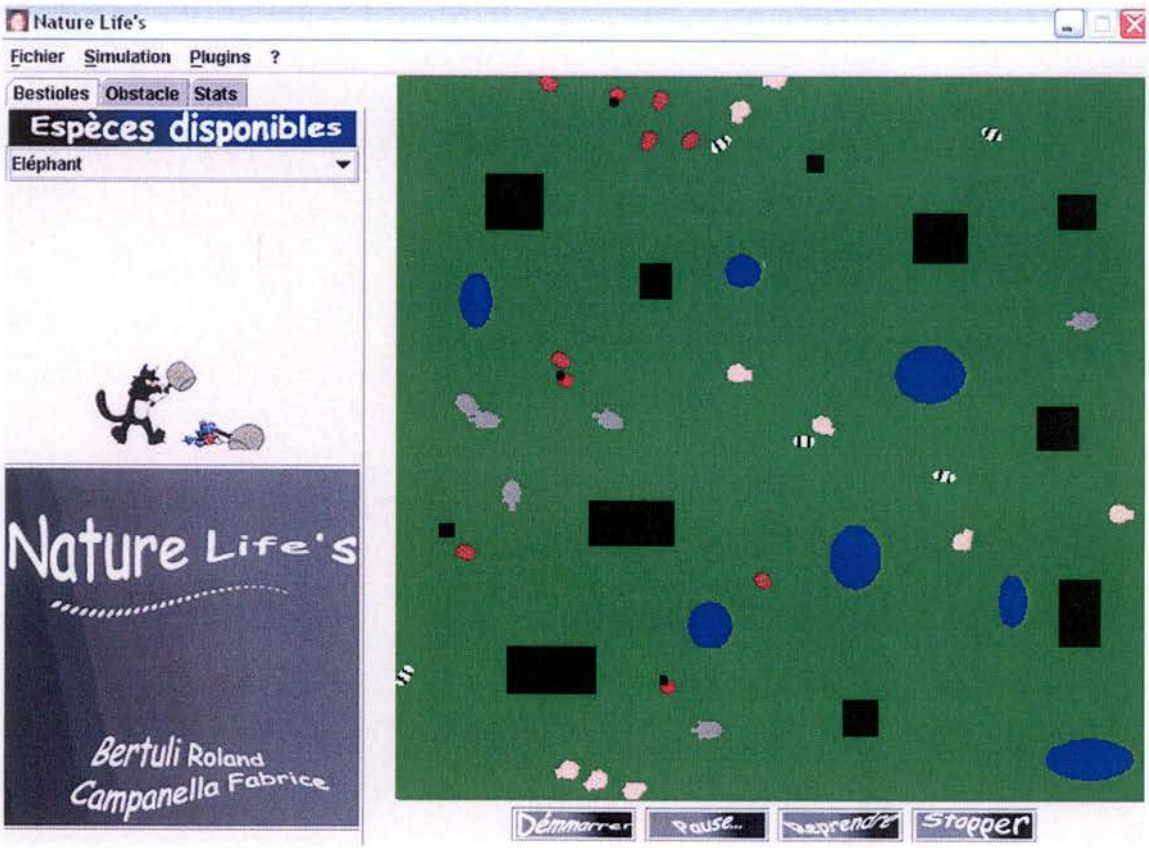


FIG. 1.1 – La fenêtre principale de l'interface, le terrain de jeu

Les espèces

Différentes espèces cohabitent dans cet univers. Elles se distinguent par quelques caractéristiques propres :

- **La distance de vue** qui détermine fortement le repérage de l'individu dans son environnement.
- **L'angle de vue** qui influence la direction dans laquelle se déplace l'animal. En effet,

la bête ne tourne que dans son angle de vision (elle ne va pas où elle ne voit pas).
- **La vitesse maximale** de déplacement.

Hormis ces caractéristiques, il faut également ajouter que l'on distingue deux types de bêtes : celles qui marchent et celles qui volent. En fonction du paramétrage du terrain, on pourra avantager l'un ou l'autre de ces deux types d'espèces. Il est également à noter que les animaux se déplaçant dans l'air volent suffisamment bas pour attraper des animaux au sol.

Les espèces constituent le premier type de plugin que l'on retrouve dans ce programme.

Les comportements

Ce sont les comportements qui donnent un caractère aux animaux et déterminent leur interaction avec l'environnement qui les entourent. Un animal peut posséder plusieurs comportements, et c'est la confrontation de ceux-ci qui détermine les réactions de la bête. Pour donner plus ou moins d'importance à l'un ou l'autre comportement, on donne un poids à chacun d'entre eux. Après chaque pas de la bête, le programme décide la nouvelle direction à prendre en fonction de l'angle de vue et de la pondération des comportements. En fait, suite à l'analyse des comportements possédés, l'application attribue elle-même un poids aux caps possibles et décide donc de la direction à prendre.

Nature Life's contient plusieurs comportements primaires que l'on peut combiner sur les bêtes :

- **Tout droit** : le comportement le plus primaire, l'animal totalement écervelé avance bêtement tout droit.
- **Ronde** : la bête est stupide et l'assume, elle tourne indéfiniment en rond.
- **Aléatoire** : l'individu n'a aucune cohérence, il avance au hasard. Il est imprévisible et n'a aucun but.
- **Ciblé** : la bête a un objectif, elle le cherche aléatoirement tant qu'elle ne le voit pas (en fait, peu importe sa distance de vue, la cible doit être dans son angle de vue). Une fois l'objectif repéré, elle y va en ligne droite. Quand elle arrive à bon port, elle s'arrête et ne fait plus rien.
- **Prudent** : la bête prudente a pour objectif d'éviter les obstacles.
- **Peureux** : l'animal, très peureux, évite comme la peste tout individu.
- **Collectif** : l'individu avance aléatoirement jusqu'à ce qu'il aperçoive un congénère. Il se sentira envahi par un formidable élan d'amitié, et le suivra. Ce comportement appliqué à plusieurs individus de la même espèce, donnera naissance à des troupeaux d'amis.
- **Prédateur** : la vie est dure et seuls les plus forts survivent. Le prédateur recherche ses proies en avançant aléatoirement. Dès qu'il en repère une, il se jette sur elle en courant. Une fois attrapée il la dévore avec fougue et s'abandonne à une sieste aussi courte que salvatrice, avant de repartir en chasse.

Les comportements correspondent au second type de plugin que l'on retrouve dans l'ap-

plication.

Implémentation

Voyons à présent, à partir du code source et d'explications livrées sur le site, comment ces différents mécanismes ont été programmés et comment le principe du plugin a ici été utilisé.

Les plugins de Nature Life's

Dans Nature Life's, les bêtes sont des classes filles de la classe abstraite `"Beast"`, qui comporte toutes les caractéristiques communes aux bêtes. Cette classe implémente l'interface `"Plugin"`. Ceci signifie que les bêtes sont des plugins que peut créer n'importe quel programmeur courageux. Cependant Nature Life's est à la base fourni avec cinq types de créatures (voir Annexe 1, page 86).

Pour être reconnues par le programme, les nouvelles espèces devront donc hériter de la classe `"Beast"`, et implémenter l'interface `"Plugin"`. Elles implémenteront obligatoirement les quatre méthodes suivantes :

- `drawYou(Graphics g)`, qui dessine la bête telle qu'elle sera vue sur l'aire de jeu
- `wherePhoto()`, qui indique le nom du fichier contenant l'image de la bête
- `getName()`
- `getAbout()`

Les deux premières sont les deux méthodes abstraites de la classe `"Beast"`. Elles doivent obligatoirement être implémentées dans toute classe qui définit une nouvelle espèce. Si les autres méthodes de la classe `"Beast"` qui donnent les caractéristiques propres à l'espèce ne sont pas redéfinies, les champs qui correspondent à ces caractéristiques se verront attribuer des valeurs particulières par défaut. Notons qu'il n'y a dans ce cas pas vraiment d'intérêt à développer une nouvelle espèce. Les deux dernières sont les seules méthodes de l'interface `"Plugin"`, elles renvoient des chaînes de caractères correspondant au nom et à quelques informations sur le plugin. Un exemple d'implémentation d'une espèce est fourni en Annexe 2 (page 88).

En ce qui concerne les comportements, également programmés sous forme de plugins, ils correspondent à des classes filles de la classe abstraite `"Behaviour"`. Celle-ci comporte toutes les caractéristiques communes aux comportements. Cette classe implémente l'interface `"Plugin"`. Rappelons qu'un certain nombre de comportements de base (voir section précédente) est compris dans la version disponible de l'application.

Les nouveaux comportements devront donc hériter de la classe `"Behaviour"`, et implémenter l'interface `"Plugin"`. Ils redéfiniront obligatoirement les trois méthodes ci dessous :

- `giveCourses()`
- `getName()`
- `getAbout()`

La méthode "`givecourses`" sera appelée après chaque pas de l'animal. Elle calcule les poids attribués aux différents caps que peut prendre l'animal et influence donc directement le déplacement de ce dernier. Elle renvoie un tableau de valeur qui correspond précisément à l'ensemble de ces poids. Le principe de fonctionnement de la pondération des caps est expliqué sur la page du site dédiée aux comportements [7]. Les deux dernières sont les deux méthodes de l'interface texttt"Plugin", déjà expliquées précédemment. Un exemple de définition d'un comportement se trouve en Annexe 2 (page 88).

Soulignons ensuite que, lors du chargement d'un comportement par l'application, cette dernière appelle la méthode "`initBehaviour(Area a, Beast b, int i)`" de la classe "`Behaviour`". Celle-ci lie le nouveau comportement à la bête qui va le posséder et à l'instance du terrain sur lequel se trouve l'animal. L'entier correspond à un poids que choisit le programmeur, qui déterminera le comportement "dominant" en cas de conflit entre comportements.

Précisons également que le même genre de technique est utilisé avec les classes correspondant à des espèces. Ce mécanisme permet par ailleurs d'éviter la nécessité d'un constructeur pour les définitions de plugins.

Cette technique est très intéressante dans l'étude qui nous concerne car c'est grâce à cette dernière que les plugins communiquent avec le noyau de l'application principale, et ce de manière transparente. En effet, le programme principal sait juste que lorsqu'il charge un plugin, il doit simplement appeler la méthode d'initialisation de ce dernier sans savoir ce qu'il y a derrière celle-ci. Cette méthode se servira des informations données par le programmeur dans la définition du plugin, qui donnent les caractéristiques spécifiques du plugin en question. Parallèlement à cela, le concepteur du plugin connaît simplement les méthodes qu'il doit (re)définir pour qu'il fonctionne correctement. Il n'est nullement obligé de savoir comment fonctionne le reste du programme.

Diagramme des classes Java

Voyons à présent graphiquement les différentes classes qui interviennent dans le mécanisme du plugin, ainsi que leurs relations. Les différents symboles utilisés dans la figure 1.2, qui montrent ces classes, sont à comprendre comme suit :

- * : Les différents attributs des classes respectives ont été regroupés sous le symbole {attributs} puisqu'ils ne nous intéressent pas directement.
- ** : Le symbole {sets} regroupe les méthodes des classes `Beast` et `Behaviour` qui ont

[7] http ://rolab.free.fr/Project/Maitrise/NatureLife/comportements.htm

pour but d'affecter des valeurs aux différents attributs de ces deux classes. Elles sont en principe utilisées par les développeurs pour caractériser les plugins qu'ils conçoivent.

- *** : Le symbole {autres} correspond pour chaque classe respective à toutes les autres méthodes définies. Il s'agit souvent de méthodes "intermédiaires" ajoutées par le développeur de plugins.
- Les deux classes encadrées en pointillés font référence à des classes génériques qui correspondent à des implémentations de plugins. Elles sont mises en évidence car elles ne font pas partie de la "statique" du système mais coïncident avec des entités appelées dynamiquement. Si nous devions faire une analyse de l'état du programme à un moment donné de son exécution, nous aurions donc probablement plusieurs instances de ce type de classes. Remarquons-y simplement la présence des méthodes qui doivent obligatoirement être implémentées (voir section précédente) pour assurer le bon fonctionnement des plugins.

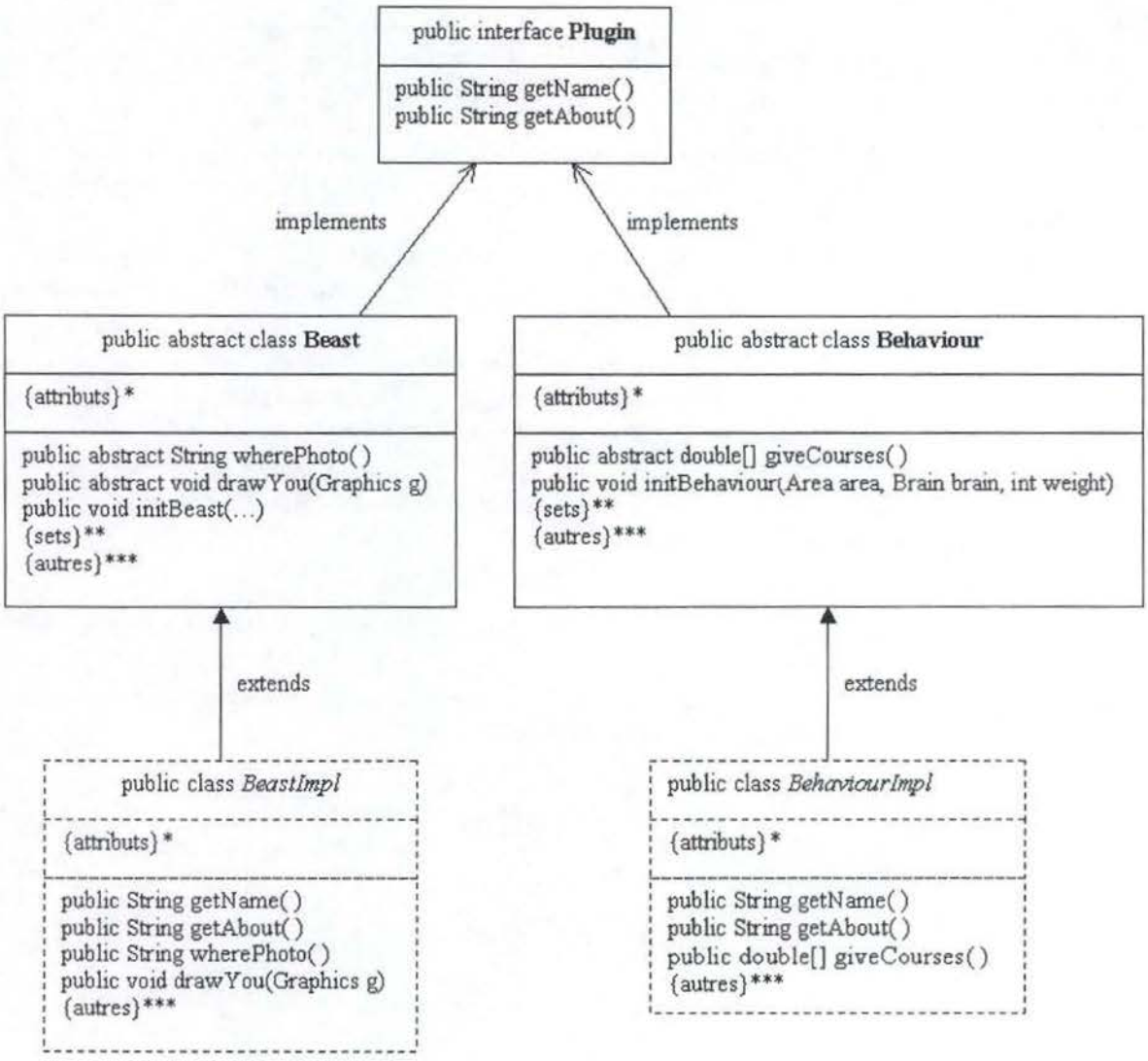


FIG. 1.2 – Diagramme des classes Java intervenant dans le mécanisme du plugin

Découpe en packages

Dans les logiciels orientés-objet, on peut considérer que la découpe en packages est souvent représentative des différents composants, choisis au cours d'une étape antérieure du développement. L'analyse future de cette découpe pourrait donc être bien utile en vue de l'élaboration d'une architecture "générique" permettant de mettre aisément en oeuvre le principe du plugin. Dans cette optique, la figure 1.3 représente l'architecture de Nature Life's. Chaque encadré y correspond à un package. Expliquons brièvement les fonctions qui leurs sont associées, package par package, afin de mieux cerner les rôles des différents composants :

- **area** : regroupe les classes qui définissent le terrain de jeu et ses propriétés, ainsi que celles qui implémentent le chargement et la sauvegarde de configurations particulières du terrain.
- **intelligence** : contient la définition du "cerveau" de la bête. C'est ici que sont notamment définies les tactiques à adopter à chaque pas, en fonction de l'environnement et des comportements "possédés".
- **natureGUI** : définit les composants de l'interface graphique.
- **obstacle** : reprend la définition des différents types d'obstacle. Il précise également une certaine interface que doit implémenter chaque obstacle pour interagir correctement avec l'environnement et les animaux.
- **pluginSDK** : regroupe les classes qui définissent les plugins de NatureLife's, et qui doivent être (partiellement) connues du programmeur de plugins.
- **plugins** : gère le chargement des plugins à partir des répertoires.

Remarquons encore que la classe esseulée "NatureLife" est la classe mère permettant de lancer l'application. Le package "bertuli.util", lui, est utilisé lors du calcul d'angles et de distances, et pour réaliser des tirages aux sort. Il n'est donc pas essentiel pour comprendre l'architecture du logiciel et pourra être omis dans l'analyse future.

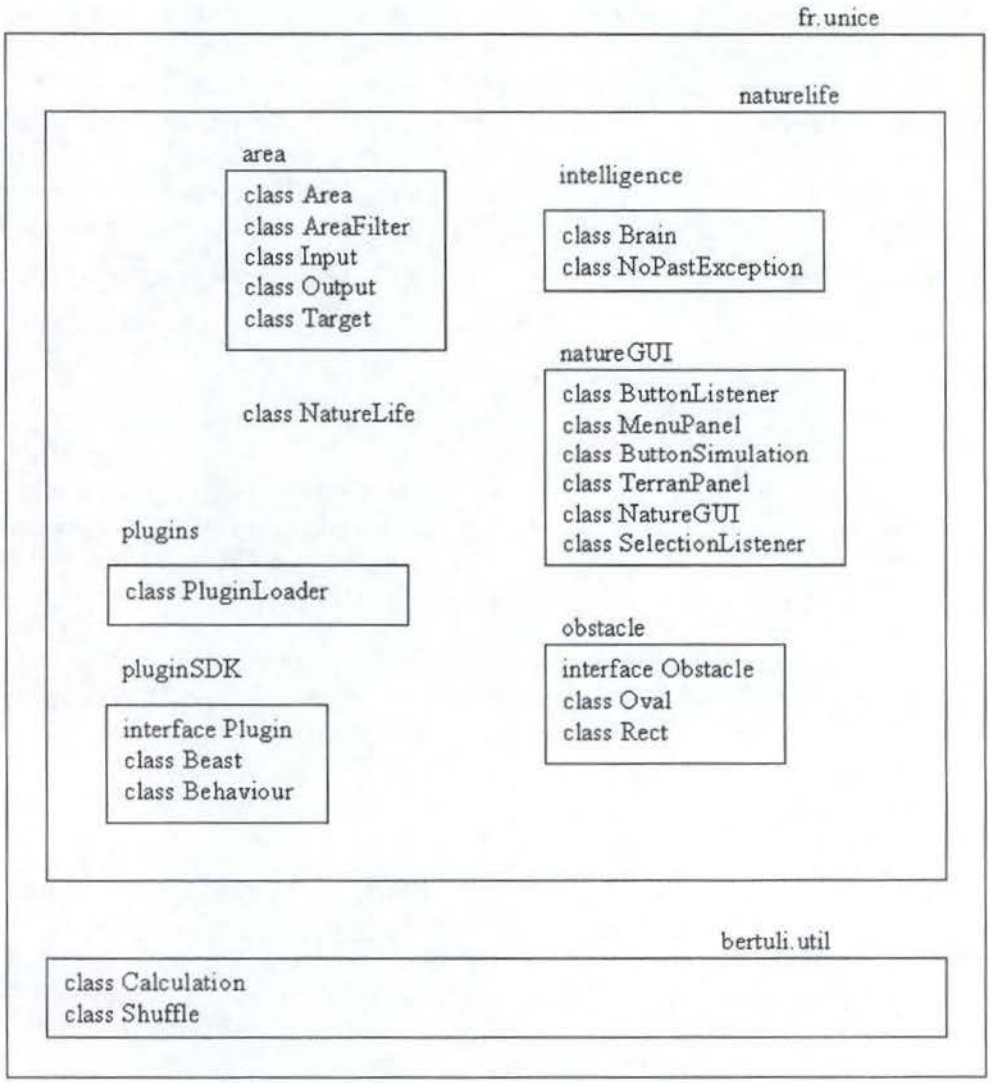


FIG. 1.3 – Découpe en packages de Nature Life's

Principe de reconnaissance et de chargement des plugins

Pour être reconnues par l'application, les nouvelles définitions d'espèces et de comportements, auparavant compilées à l'aide d'une partie des sources de l'application, doivent être placées respectivement dans deux répertoires bien déterminés. De cette façon, lors du désir de l'utilisateur de charger une de celles-ci, le programme sait directement où il doit aller voir pour charger les plugins.

Sans trop entrer dans des détails techniques, notons qu'il existe en Java, comme dans

beaucoup d'autres langages, des mécanismes permettant d'instancier des classes "au vol". Lorsque l'on dispose du nom de la classe que l'on désire instancier, on peut par exemple utiliser la méthode statique "`forName(String classname)`" de la classe "`Class`" en lui passant ce nom en argument. Celle-ci nous renvoie un élément de type "`Class`" auquel on peut appliquer la méthode "newInstance()" et ainsi récupérer l'instance dont on a besoin. Notons enfin que le mécanisme de "type casting" permet de donner le type qui convient à l'objet ainsi récupéré. Le code source de la classe "`PluginLoader`" qui emploie notamment cette technique est fourni en Annexe 3 (page 93).

1.4.2 ImageJ

ImageJ est un programme de domaine public de traitement d'image. Il est inspiré par "NIH Image" pour les Macintosh. Il fonctionne aussi bien en temps qu'applet en ligne que comme application téléchargeable, et ce sur n'importe quel ordinateur possédant une machine virtuelle Java 1.1, ou postérieure. L'auteur en est Wayne Rasband (*Research Service Branch, National Institute of Mental Health, Bethesda, Maryland, USA*).

Toutes les informations données sur ImageJ dans ce chapitre peuvent être trouvées à partir du site de l'application [8] et sur le tutorial écrit par Werner Bailer [Bai01].

Description d'ImageJ

Avec ImageJ, il est possible d'afficher, éditer, analyser, traiter et sauvegarder des images codées sur 8 bits, 16 bits et 32 bits. On peut également lire des images de différents formats, tels que TIFF, GIF, JPEG, BMP, DICOM, FITS, ainsi que des images "brutes" [9]. ImageJ supporte également l'ouverture et la lecture de piles, c'est-à-dire des séries d'images partageant la même fenêtre d'affichage. Voici quelques fonctionnalités proposées dans la version "de base" d'ImageJ :

- Mesure de distances et d'angles
- Création d'histogrammes de densité
- Manipulation du contraste
- Traitement d'images (manipulation du contraste, détection de contours, affilage et lissage d'images)
- Transformation géométrique (changement d'échelle, rotation, renversement d'images)
- Zoom agrandissant jusque 32 :1 et zoom rétrécissant jusque 1 :32
- etc ...

[8] http ://rsb.info.nih.gov/ij/

[9] c'est-à-dire de simples tableaux de pixels, sans aucun mécanisme de compression, ni structure de données additionnelle

Quelques unes des ces fonctionnalités sont présentées graphiquement en Annexe 4 (page 97).

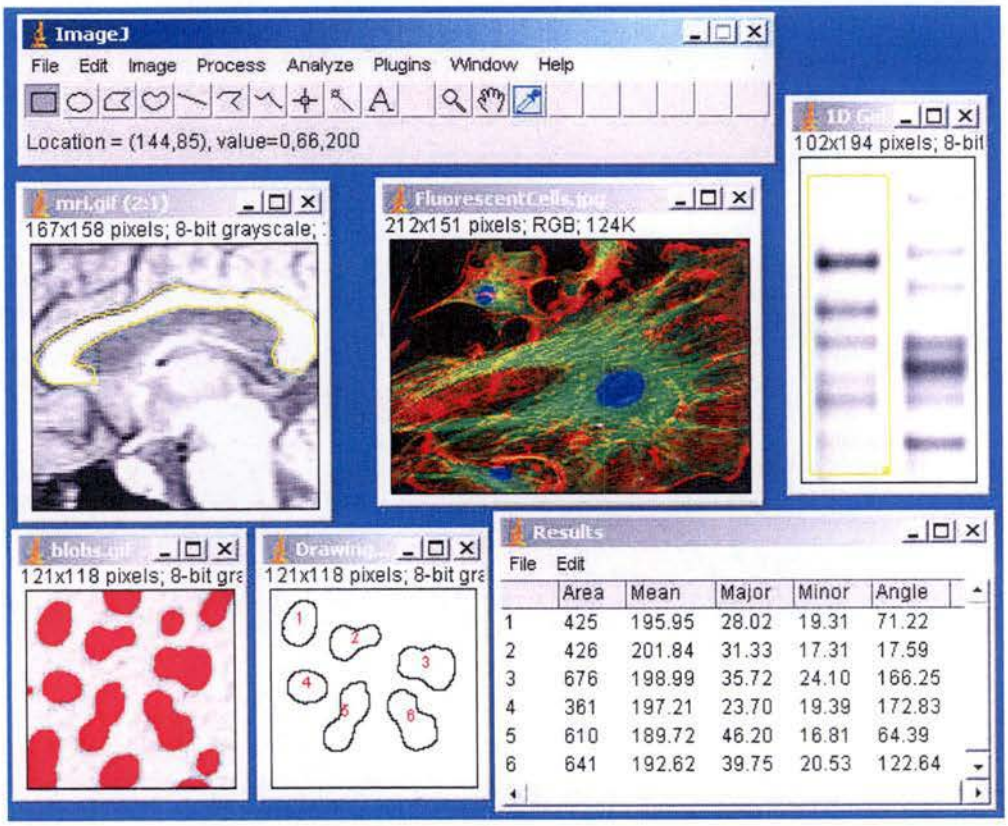


FIG. 1.4 – Interface graphique d'ImageJ

Comme le montre la figure 1.4, la fenêtre principale d'ImageJ contient une barre de menus, une barre d'outils, un champs affichant le statut de l'image sélectionnée et une barre de progression de l'action en cours. Les images, les histogrammes, et les autres données visualisables sont toujours affichées dans des fenêtres supplémentaires. Les résultats des mesures apparaissent dans une fenêtre "Results".

L'architecture ouverte d'ImageJ offre une certaine extensibilité du programme, grâce au mécanisme de plugin Java. Ceux-ci peuvent être développés par des tiers et offrir de nouvelles fonctions d'analyse, d'acquisition ou de traitement. Ils peuvent d'ailleurs résoudre la plupart des problèmes d'analyse ou de traitement d'images. Tout qui souhaite implémenter un plugin pour ImageJ devra se plier à certaines obligations de programmation, sans pour autant être obligé d'étudier la totalité des sources du logiciel. Les contraintes à respecter et les principales parties de l'application dont il est bon de connaître le fonctionnement pour écrire un plugin valable, sont expliquées en détail dans un manuel d'instruction écrit par un membre de l'équipe de développement [Bai01].

La démarche à suivre pour rendre un plugin exécutable dans ImageJ n'est pas compliquée. Après l'avoir écrit il suffit d'invoquer l'item "Compile and Run ... " dans le menu "Plugins" (1.5) et sélectionner le fichier ".java" correspondant. ImageJ se charge alors de la compilation (ou feedback en cas d'erreur) et place le fichier ".class" correspondant dans ce même répertoire. Un item permettant d'exécuter le nouvel outil est alors inséré dans le menu "Plugins" .

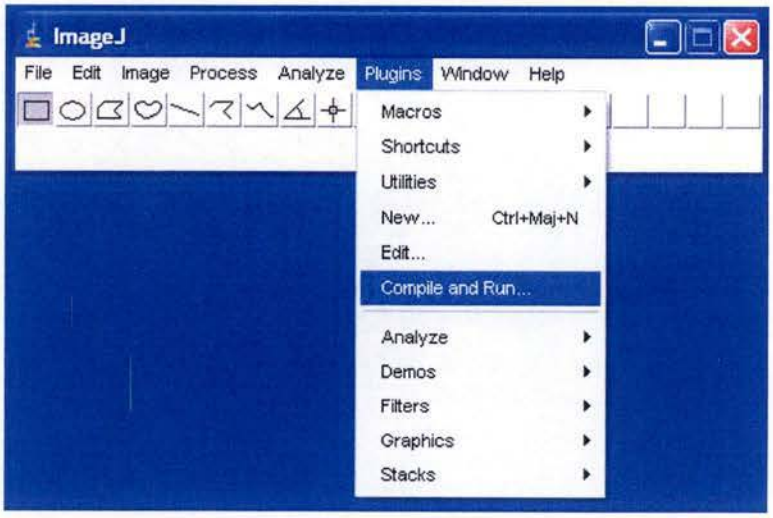


FIG. 1.5 – Le menu "Plugins" d'ImageJ

Il existe dans ImageJ deux types de plugins. En effet, on différencie ceux qui traitent une image qui doit être fournie en entrée, et les autres, qui ne "travaillent" pas sur une image particulière. Les premiers correspondent généralement à des filtres ou à des analyseurs d'image. Nous les appellerons dorénavant "plugins filtres". Les autres, eux, définissent de nouvelles extensions, telles que des supports pour traitement de nouveaux formats d'image, par exemple.

Notons qu'ImageJ présente une certaine originalité, dans le sens ou un grand nombre d'outils fournis avec le logiciel de base sont en fait eux-mêmes des plugins. Ils se différencient des plugins définis par des tiers par l'endroit où ils sont stockés dans la structure de l'application, mais fonctionnent de la même manière que ceux-ci.

Implémentation

Voyons comment est implémenté le principe de plugin dans cette application. Comme pour Nature Life's, nous disposons, pour commencer notre analyse, du code source de l'application et de quelques explications le concernant.

Les plugins d'ImageJ

Les plugins d'ImageJ sont en fait des classes Java qui implémentent les interfaces nécessaires et sont placées dans un répertoire particulier. Les définitions de plugins filtres doivent implémenter l'interface "`PlugInFilter`". Le second type, lui, implémente l'interface "`PlugIn`". Les codes sources de ces deux interfaces sont donnés en Annexe 5 (page 100), et deux exemples d'implémentation de plugins sont fournis en Annexe 6 (page 102).

L'interface `PlugIn` contient une méthode unique dont la signature est `void run(java.lang.String arg)`. Cette méthode "lance" le plugin effectivement. Ce qu'on y définit dans les implémentations est ce que le plugin réalise effectivement. Tout le code du plugin se situe donc dans cette méthode, ou est appelé à partir de cette dernière. `arg` est une chaîne de caractères, pouvant être vide, passée en argument du plugin. Chaque plugin peut être installé plusieurs fois, auquel cas chacun d'entre eux appellera la même classe, mais avec un argument différent.

L'interface `PlugInFilter` possède également une méthode "run" de signature `void run(ImageProcessor ip)` qui, dans les implémentations effectives, définit également tout ce que le plugin fait réellement. Notons que dans ImageJ, une image est définie par la classe `ImagePlus`. On traite une image à l'aide d'un "processeur" qui lui est lié, et qui fournit un certain nombre de méthodes permettant de travailler effectivement sur l'image. Ce processeur est défini par la classe `ImageProcessor`. Remarquons que la classe ImagePlus possède en réalité un attribut `protected ImageProcessor ip`. Le processeur de l'image que l'on désire modifier est donc passé en argument de la méthode `run`. Ce processeur peut être modifié directement. Cependant, une nouvelle "`ImagePlus`" initialisée avec les mêmes données que celle de départ peut être créée (avec un nouveau processeur). Dans ce cas, l'image originale reste inchangée, et une nouvelle image subissant les modifications est alors construite. Contrairement à son homonyme de l'interface `PlugIn`, la méthode `run` ne prend pas de chaîne de caractères en argument. Cet argument peut en fait être passé grâce à la seconde méthode de l'interface `PlugIn`, de signature `int setup(String arg, ImagePlus imp)`. Cette dernière est appelée automatiquement une fois le plugin filtre chargé par l'application. Son rôle est en quelque sorte de "préparer" le filtre avant son utilisation. L'argument `arg` y a la même fonction que dans l'interface `PlugIn`. Le programmeur ne doit pas se soucier de l'argument `imp` qui est géré par ImageJ, et qui désignera l'`ImagePlus` représentant l'image courante. L'entier renvoyé par la méthode `setup` indique le type d'image que le filtre traite (par exemple des images couleur codées sur 8 bits ou des piles d'images). Les relations entre les entiers à renvoyer et les types d'image sont définies dans l'interface `PlugInFilter`.

Il ressort donc de ce qui vient d'être dit que la seule contrainte qui est imposée au programmeur de plugin est l'implémentation d'une des deux interfaces présentées juste avant. Il a donc, au plus, deux méthodes à définir, pour que son plugin soit pris en charge par l'application. Comme dans Nature Life's, tout se passe donc de manière transparente lorsqu'un plugin est appelé. En effet, le programme principal invoque automatiquement la méthode `run` qui existe forcément, puisque un plugin valable implémente d'office cette méthode. La

communication avec les composants intéressants de l'application se fait via l'utilisation des quelques classes définies dans le manuel d'instructions. Ces dernières jouent en réalité le rôle d'interface de communication entre le noyau du logiciel principal et les plugins. Notons qu'elles constituent la seule partie de l'application que les programmeurs tiers doivent connaître un minimum.

Remarquons simplement, pour terminer cette section, que les plugins d'ImageJ s'exécutent dans des *Threads* différents. Par conséquent, des opérations consommant beaucoup de temps processeur peuvent s'exécuter en parallèle avec d'autres opérations.

Diagramme des classes Java

Comme auparavant, représentons graphiquement les classes expliquées ci-dessus qui interviennent dans le mécanisme du plugin, ainsi que leurs relations.

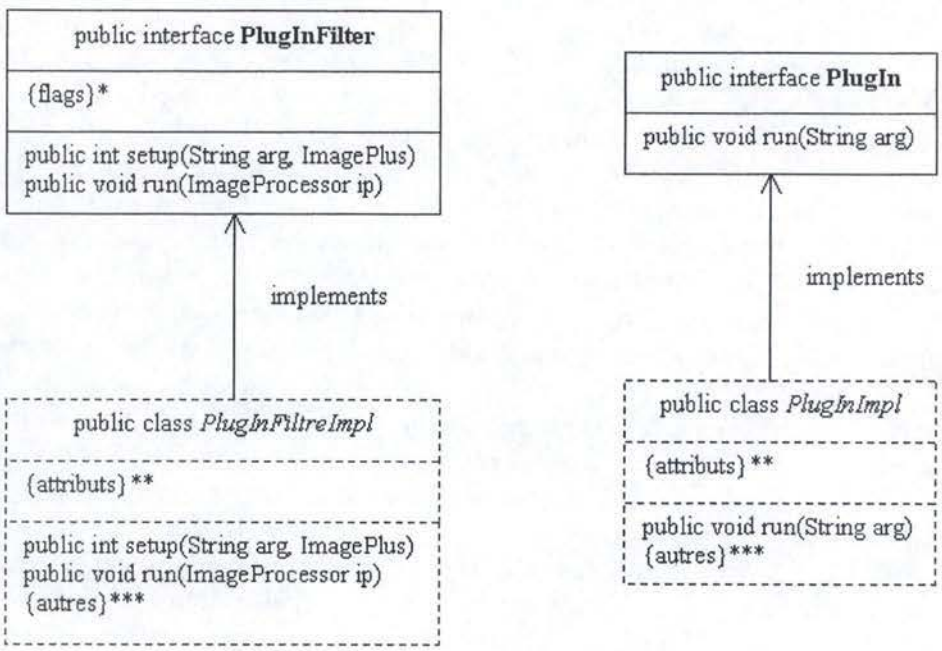


FIG. 1.6 – Diagramme des classes Java intervenant dans le mécanisme du plugin

Les symboles particuliers de la figure 1.6 sont à comprendre comme suit :

- * : On regroupe ici la définition des flags désignant les différents types d'images.
- ** : Les différents attributs des classes d'implémentations de plugins ont été regroupés sous le symbole {attributs} puisqu'ils ne nous intéressent pas directement.

- *** : Le symbole {autres} correspond pour chaque classe respective à toutes les autres méthodes définies. Il s'agit souvent de méthodes "intermédiaires" ajoutées par le développeur de plugins.
- Comme pour Nature Life's, les deux classes encadrées en pointillés font référence à des classes génériques qui correspondent à des implémentations de plugins. Elles ne font pas partie de la "statique" du système mais coïncident avec des entités appelées dynamiquement. Une analyse de l'état du programme à un moment donné de son exécution nous donnerait donc probablement plusieurs instances de ce type de classes. Remarquons encore la présence des méthodes qui doivent obligatoirement être implémentées (voir section précédente) pour assurer le bon fonctionnement des plugins.

Découpe en packages

Pour les mêmes raisons que précédemment avec Nature Life's, nous allons ici présenter la découpe en packages d'ImageJ. On considérera à nouveau que celle-ci est relativement représentative des différents composants définis plus tôt dans le développement du logiciel. Rappelons que le but de cette analyse est l'élaboration d'une architecture "générique" permettant de mettre facilement en oeuvre le principe du plugin.

La figure 1.7 représente cette découpe. Chaque encadré y correspond à un package. Le symbole "{classes}" désigne simplement un ensemble de classes qui n'ont pas un grand intérêt dans l'analyse qui nous concerne. Expliquons à nouveau les fonctions qui leurs sont associées, package par package, en vue de comprendre les rôles des différents composants :

- **ij** : est le package de base reprenant l'entièreté du programme. Hormis les packages qui vont être commentés ci-dessous, on y trouve notamment quelques classes essentielles au fonctionnement de l'application. Parmi celles-ci remarquons les classes qui définissent les représentations des images et des piles d'images dans ImageJ. S'y situent également la classe `IJ` qui lance l'exécution des différentes commandes (notamment l'exécution des plugin's), et `ImageJ`, la classe mère de l'application permettant de lancer cette dernière.
- **plugin** : regroupe les classes qui définissent les plugins d'ImageJ. C'est principalement le contenu de ce package qui doit être connu du programmeur de plugins. Le symbole "{plugin classes}" représente un ensemble de définitions de plugins simples, c'est-à-dire implémentant l'interface `PlugIn`.
- **filter** : comprend les classes qui définissent les plugins filtres d'ImageJ, avec principalement l'interface `PlugInFilter`. Le symbole "{plugin filter classes}" représente un ensemble de définitions de plugins filtres, c'est-à-dire implémentant l'interface `PlugInFilter`.
- **frame** : définit les différents composants et la classe principale d'une fenêtre que les plugins peuvent étendre pour afficher un résultat quelconque.
- **gui** : définit les composants de l'interface graphique de l'application.
- **io** : définit le composant qui gère les entrées/sorties du programme. Il s'occupe ainsi de la lecture (le décodage) et de l'écriture (l'encodage) des différents fichiers images. Il se consacre également, via la classe `PlugInClassLoader`, au chargement des plugins, à

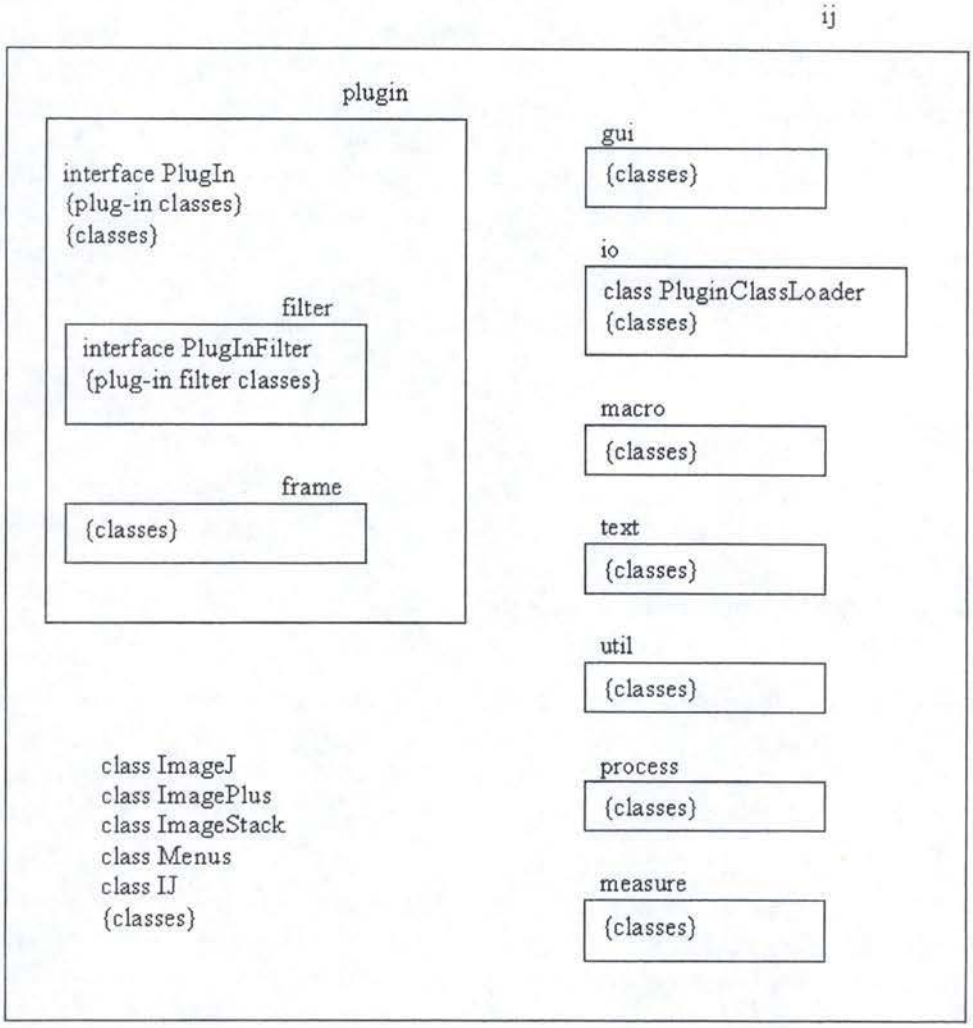


FIG. 1.7 – Découpe en packages d'ImageJ

partir du répertoire qui leur est destiné. Nous reviendrons dans la section suivante sur l'utilité de cette classe.

- **macro** : comprend les classes qui gèrent des fonctions "macros" pouvant être écrites sous forme de fichier texte et gérées par ImageJ.
- **process** : reprend les classes qui permettent de traiter les différents types d'images. On y trouve notamment la super-classe abstraite `ImageProcessor` que nous avons évoquée précédemment.
- **measure** : contient les classes nécessaires pour effectuer un certain nombre de mesures (angles, distances, ...) sur les images affichées.
- **text** : fournit les outils nécessaires à l'affichage de texte dans une fenêtre de l'applica-

tion.
- **util** : regroupe un certain nombre d'outils utilisés par plusieurs autres composants.

Principe de reconnaissance et de chargement des plugins

Nous avons dit auparavant que beaucoup de fonctionnalités offertes par ImageJ étaient implémentées sous la forme de plugins. Ces derniers font cependant partie du code source de départ de l'application, et ne rentrent pas exactement dans la définition que nous avons donnée dans la section "Terminologie". Par conséquent nous traiterons dans cette section uniquement les plugins écrit par des tiers, qui sont chargés dynamiquement et correspondent bien avec la définition proposée.

Le principe de reconnaissance des plugins est quasi identique à celui de Nature Life's. En effet, les sources des plugins doivent être placées dans un répertoire particulier, où le programme ira voir lors du souhait de l'utilisateur d'en charger un. Une petite subtilité est cependant utilisée dans ImageJ, pour différencier les fichiers correspondant à des définitions de plugin's et d'éventuels autres fichiers qui pourraient se situer dans le même répertoire. Il est en fait demandé au programmeur tiers que le nom du fichier plugin se termine par "_". De cette façon, lors de la recherche des plugins par le programme, ce dernier effectue un filtrage pour être sûr de chercher parmi les bonnes sources. De plus, le nom que prendra le plugin dans le menu de l'interface graphique est directement déterminé par le nom du fichier qui lui correspond [10].

Regardons à présent les quelques instructions-clés qui permettent le chargement et l'exécution des plugins d'ImageJ. La méthode `runUserPlugIn` de la classe `IJ` contient ces instructions. Dans un premier temps on récupère un objet de type `"Class"` grâce à l'instruction `loader.loadClass(className)`, où `loader` est une instance de `PluginClassLoader`, situé dans le package `io`. Pour une question d'efficacité, `PluginClassLoader` étend en réalité la classe java `ClassLoader` et redéfinit la méthode `loadClass`, qui permet de charger une classe dynamiquement dans un programme java. L'argument `className` est une chaîne de caractères correspondant au nom du fichier ".class" du plugin concerné. Comme dans Nature Life's, on crée alors un objet qui est une instance de cette classe, via la méthode `newInstance()`. Une fois l'instance récupérée, on peut encore vérifier s'il s'agit d'une implémentation de l'interface `PlugIn` ou de l'interface `PlugInFilter`, grâce à l'instruction `instanceOf` qui renvoie une valeur booléenne. On utilise alors le mécanisme de "type casting" pour donner le bon type à l'instance du plugin. Il reste alors à lancer l'exécution du plugin en appelant sa méthode `run`, commentée auparavant. Pour plus de détails sur ces instructions, on se référera à l'Annexe 7 (page 104) qui fournit les parties de codes qui nous intéressent dans les fichiers "PluginClassLoader.java" et "IJ.java".

[10] Le programme enlève en fait le dernier "_" et remplace les autres par un espace. Par exemple, un plugin qui doit être appelé "Color Inverter" devra être défini dans un fichier de nom "Color_Inverter_.java ."

1.4.3 Router Plugins

Router Plugins est une architecture logicielle extensible fournissant des services de routage très performants. Elle a été créée par l'équipe en charge du projet du même nom, composée de chercheurs du *Computer Engineering and Networks Laboratory* à l'*ETH Zurich* [11] et du *Applied Research Laboratory* de l'université de Washington [12]. Cette architecture a été conçue pour être intégrée dans le système d'exploitation NetBSD [13]. Toutes les informations dont nous disposons proviennent d'un article rédigé par des chercheurs membres du projet [DDPP98].

L'architecture de Router Plugins permet à des modules, appelés plugins, d'être dynamiquement chargés dans le noyau du logiciel et configurés en cours d'exécution de celui-ci. Ils sont par ailleurs assignés à une fonction spécifique sur un flux particulier traversant le réseau. Nous considérerons donc que ces modules correspondent avec la définition que nous avons donnée dans la section "Terminologie".

Intérêt de l'architecture modulaire et de l'utilisation de plugins

Le nombre de protocoles réseau et d'extensions à ceux-ci ne cessent d'augmenter depuis l'utilisation toujours grandissante des réseaux de tous types. Parallèlement à cela, on voit également apparaître de plus en plus de nouvelles fonctionnalités proposées par les logiciels de routage. Dans le passé, la tâche principale d'un routeur était simplement de transférer des paquets (ou datagrammes) sur base de la vérification d'une adresse de destination. Les routeurs modernes, eux, proposent un certain nombre de nouveaux services supplémentaires qui peuvent être intégrés :

- améliorations des techniques classiques de routage, avec choix de la technique à utiliser
- algorithmes de sécurité ajoutés au routage simple (permettant par exemple d'implémenter des réseaux privés virtuels)
- améliorations des protocoles existants
- nouveaux protocoles de base (par exemple le protocole IPv6 [DH95] [14], dernière version en date d'IP).

L'architecture modulaire proposée ici, dite de type "EISR" (pour *Extended Integrated Services Router*), est donc très différente d'une architecture classique de logiciel de routage simple, désignée par le terme *Monolithic Best-Effort Architecture*. La figure 1.8 compare ces deux types de découpes en composants.

[11] http ://www.tik.ee.ethz.ch/
[12] http ://www.arl.wustl.edu/
[13] http ://www.netbsd.org
[14] Même si la dernière version du protocole IP est la version 6, la transition entre la version précédente (IPv4) et la dernière est relativement lente. C'est pourquoi beaucoup de dispositifs fonctionnent encore avec les seuls mécanismes de la version 4

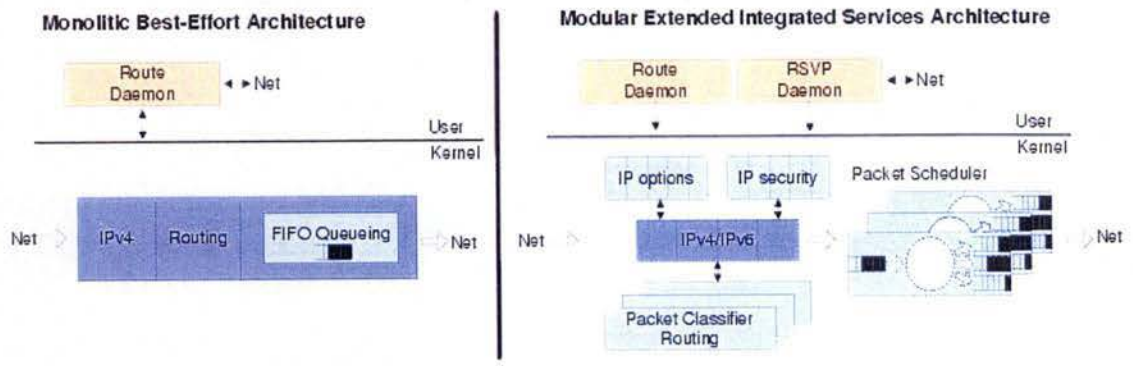


FIG. 1.8 – Architecture de routage "Best effort" et Architecture modulaire étendue avec services intégrés (*EISR*)

Sans trop entrer dans les détails, remarquons que dans la première découpe, tous les services de routage sont fournis par un même composant. Par contre, dans une architecture "*EISR*", on voit apparaître différents composants supplémentaires assignés à des tâches différentes, correspondant aux services proposés par les routeurs modernes. Ces composants sont :

- un "*sheduler*" qui organise le traitement des différents types de paquets et des divers flux qui traversent le routeur
- un "*classifier*" qui classe les différents types de paquets suivant le flux auquel ils appartiennent [15]
- des mécanismes de sécurité
- des mécanismes d'analyse des différentes options modulables dans le protocole IP.

Chacun de ces composants peut être implémenté de différentes façons et utiliser des algorithmes qui procurent des avantages spécifiques, notamment en terme de performance et de coût. La plupart de ces algorithmes subissent une constante évolution et sont fréquemment remplacés ou mis à jour. C'est ce phénomène qui amène à faire la distinction entre la partie du système dont le code reste relativement stable, appelée le noyau (*core* dans la littérature anglaise [16]), et celle dont nous dirons que l'implémentation est fluide, tant elle varie de manière constante. La partie stable est principalement responsable de l'interaction avec les dispositifs matériels du réseau et la distribution des packets aux modules spécifiques. En ce qui concerne

[15] Pour comprendre la notion de flux, il suffit de se rendre compte qu'un paquet IP ne contient en général qu'une partie de l'ensemble des informations envoyées d'une entité réseau à une autre. Ceci est d'ailleurs d'autant plus vrai avec l'utilisation croissante des sources de données et des applications multimédias qui utilise de plus en plus la technique de *streaming*.

[16] On fera attention à ne pas confondre *core* et *kernel* qui signifient tout deux "noyau". En effet, alors que le *core* fait référence à la partie du logiciel qui reste plus ou moins stable, le *kernel* peut très bien évoluer dynamiquement pendant l'exécution du système. Un plugin chargé est d'ailleurs ajouté au *kernel* du système. Pour cette raison nous utiliserons le vocabulaire *core* et *kernel* dans la suite de cette section afin de bien différencier les deux.

la partie qui se situe en dehors du noyau, il est intéressant que plusieurs implémentations des composants du *EISR* puissent co-exister. Par exemple, on peut très bien désirer utiliser un type de "*scheduler*" sur une interface réseau, et un autre type sur une autre interface du dispositif. C'est précisément pour cette raison que l'utilisation de plugins s'avère très intéressante dans ce cas.

La structure de type *EISR* expliquée auparavant devra donc être implémentée de façon à répondre aux exigences expliquées dans le paragraphe précédent. Les buts de la nouvelle architecture développée deviennent alors :

- la **modularité** : l'implémentation des algorithmes spécifiques se fait sous la forme de modules appelés plugins.
- l'**extensibilité** : les nouveaux plugins peuvent être dynamiquement chargés pendant l'exécution du logiciel.
- la **flexibilité** : les instances de plugins peuvent être créées, configurées et liées à des flux de trafic spécifiques.
- la **performance** : le système doit fournir un "chemin" de données très efficace, sans introduire la nécessité de recopier des données et sans générer plus d'interruptions entre les différents processus du système. Par ailleurs l'utilisation d'une structure modulaire ne doit pas avoir d'impact sérieux sur les performances.

Remarquons à ce sujet, qu'en comparaison avec un noyau classique de type "Best-effort", l'implémentation de l'architecture expliquée ci-après ne nécessite qu'une augmentation de 8% d'utilisation de processeur sur le type de machine employé. Ceci peut est considéré comme une performance très honorable, compte tenu du nombre de services supplémentaires proposés.

Architecture générale

Regardons dans un premier temps grâce à quels mécanismes sont atteints les buts cités ci-dessus, en introduisant déjà certains composants de l'architecture finale.

Commençons par examiner le chargement et le déchargement dynamiques de plugins dans le sous-système du système d'exploitation chargé de la gestion du réseau. Rappelons qu'un plugin est un module qui implémente une des fonctionnalités du *EISR* vues auparavant. Pour répondre à cette exigence, le système NetBSD offre un mécanisme assez puissant permettant à des modules d'être chargés dans son noyau (*kernel*). Une fois chargé, un plugin ne se différencie pas du code qui constitue le *kernel*. Cela sous-entend la nécessité d'un composant qui fait la fusion entre les plugins individuels et le sous-système de gestion du réseau. Ce composant doit également fournir une interface, dite de "contrôle", utilisée notamment par les autres éléments du noyau. Dans "router plugins", ce composant est appelé le PCU (*Plugin Control Unit*).

Parlons à présent de la création d'instances de plugins pour un maximum de flexibilité. Une instance est une configuration spécifique d'un plugin, qui peut effectivement accomplir sa

tâche durant l'exécution du programme. Remarquons qu'il est souvent préférable de pouvoir disposer de multiples instances d'un même plugin à l'intérieur du *kernel*. On peut par exemple désirer qu'une même implémentation d'un "*scheduler*" travaille avec deux configurations différentes sur deux interfaces réseau distinctes. Afin que soit fournie une interface simple et unifiée pour l'allocation de plusieurs instances d'un même plugin, les plugins devront pouvoir répondre à un ensemble de messages standardisés. En standardisant cet ensemble de messages et en les implémentant dans chaque plugin, on garantit ainsi l'inter-opérabilité parmi tous les plugins.

On a vu également qu'il était important de pouvoir associer un paquet particulier à un flux, et un flux à une instance de plugin. Les ensembles de flux sont en fait spécifiés à l'aide de structures de données appelées filtres. Par exemple, on peut imaginer un filtre qui considère tout le trafic TCP du réseau 129.0.0.0 vers l'hôte 192.94.233.10 . Un filtre peut également porter sur le flux généré par la communication entre deux applications sur deux machines différentes. Sans donner plus de détail, nous dirons simplement qu'un filtre est représenté par un six-tuple :
<adresse source, adresse destination, protocole, port source, port destination, interface d'arrivée>.
Par exemple, la spécification du filtre donné ci-dessus donnerait :
<129.*.*.* , 192.94.233.10, TCP, *, *, *>.
Nous verrons également par après que le lien entre un filtre et une instance de plugin se fait au moment où un packet IP atteint une "porte" (*gate*) au niveau de l'entité du *core* qui gère les mécanismes IP.

En ce qui concerne le désir d'atteindre de bonnes performances générales du système, nous n'entrerons pas dans les détails mais préciserons simplement que cet objectif est notamment réalisé par l'utilisation de mémoires-cache très efficaces. Nous reparlons brièvement de ceci un peu plus tard.

Examinons à présent les différents composants de l'architecture et la façon dont ils interagissent les uns avec les autres.

Les composants et le chemin de l'information de contrôle

La figure 1.9 illustre les différents composants de l'architecture et la communication dite de "contrôle" entre ceux-ci.

Décrivons à présent le rôle de "contrôle" de chacun de ces composants :

- **IPv4/IPv6 core** : le noyau (*core*) IPv4/IPv6 consiste en une implémentation des mécanismes IP des deux versions. Celui-ci contient les composants requis pour le traitement des paquets IP. Ces mécanismes restant constants, le *core* ne prend pas la forme de modules chargeables dynamiquement. Il contient principalement des fonctions d'interaction avec les dispositifs réseau et des mécanismes de distribution des paquets IP aux plugins. Comme nous allons le voir, il n'y a pas réellement d'interaction directe,

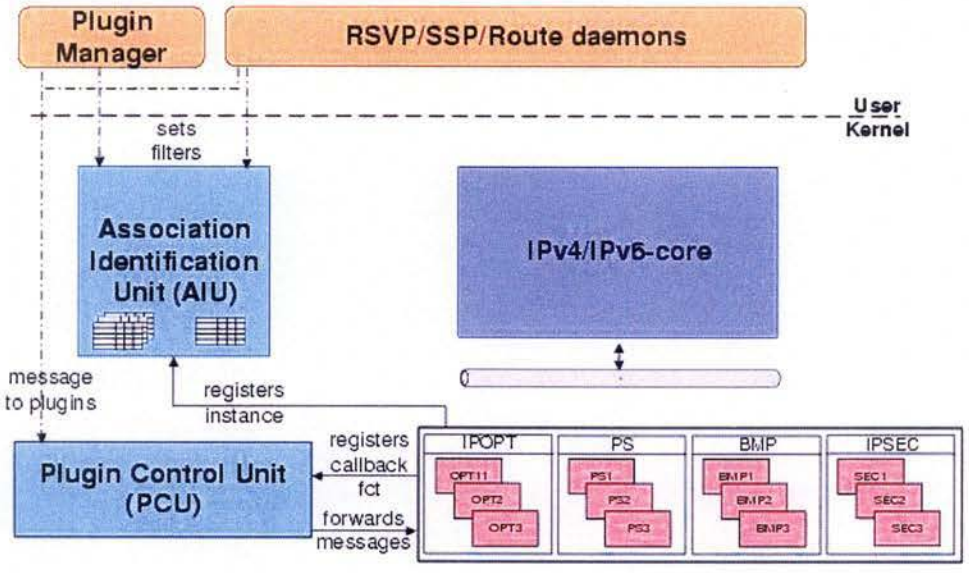


FIG. 1.9 – Architecture du système et chemin de l'information de contrôle

en terme d'information de contrôle, avec les plugins en question.

- **Plugins** : la figure 1.9 montre quatre différents types de plugins. Les premiers (*IPOPT*) implémentent les différentes options d'IPv6. D'autres (*PS*) prennent en charge ce que nous avons appelé le "*scheduling*". La troisième sorte de plugins (*BMP* pour *Best-Matching Prefix*) calcule les préfixes correspondant le mieux avec les paquets en traitement [17]. Le dernier type (*IPSEC*) met en oeuvre des mécanismes de sécurité pour IP.
- **Plugin Control Unit (PCU)** : le *PCU* gère les plugins. Il est responsable du transfert de messages aux plugins en provenance d'autres composants du noyau, ou de programmes de l'espace utilisateur, qui utilise des appels à des librairies.
- **AIU (Association Identification Unit)** : l'unité d'identification des associations implémente en réalité un "*classifier*" de paquets et réalise le lien entre les flux et les instances de plugins. Nous verrons brièvement dans la section qui suit en quoi consiste ces associations.
- **Plugin Manager** : il s'agit d'un composant qui se situe en dehors du *kernel* du logiciel, précisément dans l'espace des programmes de l'utilisateur. Il permet à ce dernier de configurer le logiciel routeur plugins de l'extérieur, grâce à l'utilisation de lignes de commandes. Le *Plugin Manager* transforme en fait ces commandes en appels à des fonctions de la librairie fournie à l'utilisateur avec le système. Précisons simplement que ces fonctions ont une portée suffisante pour permettre à l'utilisateur de configurer manuellement chacun des composants du *kernel*.

[17] Il s'agit en fait de la technique de recherche de la route la plus spécifique pour le paquet à transférer. On peut se référer à [Bon00] pour obtenir plus d'informations à ce sujet.

- **Daemons** : étant complexes à expliquer et peu importants pour la compréhension des mécanismes qui nous intéressent, nous ne retiendrons des démons que le fait qu'il s'agit de composants situés dans l'espace de l'utilisateur, qui mettent en oeuvre différents protocoles applicatifs (*RSVP* [Zea93] et *SSP* [AP] notamment).

Après un relancement du système, celui-ci doit être configuré avant d'être prêt à recevoir et à transmettre des paquets. Cette configuration suppose une sélection d'un ensemble de plugins. Etant donné que cette sélection ne s'applique pas forcément à l'ensemble des paquets traversant le routeur, il faut pouvoir fournir une définition de l'ensemble des paquets qu'une instance de plugin doit traiter. La configuration peut être réalisée par un administrateur du système, ou via l'exécution d'un script. Cette configuration implique les étapes suivantes :

- **Chargement d'un plugin** : Celui-ci se fait à l'aide de la commande "*modload*" qui fait partie de la distribution de *NetBSD*, et permet aux modules d'être chargés dans le noyau (*kernel*) du système. Au chargement, ils fournissent au *PCU* une fonction pour les rappeler (*callback function*) qui sera utilisée pour leur envoyer des messages. Il existe des messages pour créer et libérer des instances de plugins, et pour lier les instances aux flux.
- **Création d'une instance de plugin** : En utilisant l'application "*Plugin Manager*", des messages de configuration peuvent être envoyés aux plugins spécifiques. Typiquement, un message de ce type demande au plugin de créer une instance de lui-même. En cas de "*scheduling*", par exemple, l'information de configuration contiendra notamment l'interface réseau sur laquelle le plugin devra travailler.
- **Création de filtres** : Une fois qu'un plugin a été configuré et qu'une instance a été créée, il est prêt à être utilisé. Il faut encore préciser l'ensemble de datagrammes qui doit être passé à l'instance pour subir un certain traitement. Cela se fait en alliant un ou plusieurs flux à une instance de plugin. Pour spécifier l'ensemble de flux qui est supposé être manipulé par une instance particulière, le *Plugin Manager* ou un des démons peuvent créer des filtres via des appels au composant *AIU*. Rappelons que le filtre spécifie clairement un ensemble de flux.
- **Lien entre flux et instances** : Il reste alors à établir le lien entre un flux (un filtre) et une instance. Concrètement, chaque filtre dans le AIU est associé à un pointeur vers une instance. Ce pointeur est créé grâce à un autre appel au *AIU* qui fait enfin la liaison.

Les composants et le chemin des données

Bien que nous ayons décrit l'essentiel des éléments de l'architecture qui mettent directement en oeuvre le principe du plugin, il peut encore être intéressant d'examiner le chemin que suivent les données qui traversent le système. Nous ne donnerons ici que les grandes lignes de ces principes et invitons le lecteur curieux à se référer au document de base [DDPP98].

Avant de voir la séquence d'actions qui se produisent dans le système il faut brièvement

introduire la notion de porte (*gate*). De manière abstraite, nous dirons qu'une porte est en réalité un point dans le noyau IP (*IP core*) où l'exécution du système doit brancher un flux à une instance de plugin. Du point de vue du programmeur, on considérera qu'une porte est une macro qui encapsule des appels de fonctions au *AIU*. Ce dernier retourne l'instance correcte du plugin à utiliser pour traiter l'ensemble de paquets en question. Les portes sont placées dans le noyau IP là où une interaction avec un plugin doit avoir lieu. Par exemple, on peut imaginer qu'un flux qui doit subir le traitement de certaines options IP soit dirigé vers une porte responsable de ce genre de traitement. La tâche de cette porte sera alors de fournir l'instance correcte de plugin à utiliser [18].

L'*AIU* est responsable du maintien du lien entre les flux et les instances de plugin. Cela se fait grâce à l'utilisation d'une structure de données appelée table des flux (*flow table*). Cette structure est en fait une mémoire cache qui offre des accès très rapide. Comme toute cache, il existe des mécanismes d'allocation et de libération d'entrées, basés sur la fréquence de consultation de celles-ci.

Dans l'*AIU*, tous les flux commencent en dehors de la mémoire cache (*flow table*). Lorsqu'un paquet correspondant à un flux qui ne se trouve pas encore dans la table des flux arrive, la recherche dans cette table échoue. Dans ce cas, cela donne lieu à une recherche dans une autre structure de données appelée table des filtres (*filter table*). Cette dernière stocke les liens entre les filtres et les instances de plugins pour chaque porte [19]. L'algorithme de consultation de la table des filtres recherche en réalité le filtre le plus spécifique par rapport au paquet à traiter, et renvoie l'instance de plugin correspondante.

La figure 1.10 illustre ces différents mécanismes.

Terminons cette section en décrivant brièvement les étapes qu'implique le traitement d'un paquet IPv6 (voir les numéros de 1 à 6 sur la figure 1.10). Supposons que le paquet appartient à un flux non encore présent dans la table des flux et regardons la séquence d'actions qui s'ensuit :

- **0. Arrivée du paquet** : Le paquet arrive au niveau du dispositif réseau hardware qui transmet la main au noyau IP. A l'intérieur de ce dernier le paquet peut rencontrer plusieurs portes.
- **1. Rencontre d'une porte** : Supposons que la porte en question soit celle responsable du traitement des options IPv6, sa tâche est de trouver l'instance du plugin responsable du traitement des options contenues dans le paquet.
- **2. Découverte de la bonne instance** : La porte fait un appel au *AIU*. Les paramètres de cet appel sont un pointeur vers le paquet et une identification de la porte en question. Dans ce cas, nous identifierions la porte responsable des options IP comme appelante.
- **3. Classification du paquet** : Le composant *AIU* réalise dans un premier temps

[18] Remarquons qu'il y a logiquement quatre types de porte dans cette architecture, respectivement pour les quatre types de plugin (*IPOPT, PS, BMP, IPSEC*)

[19] Il y a donc logiquement quatre tables de filtres

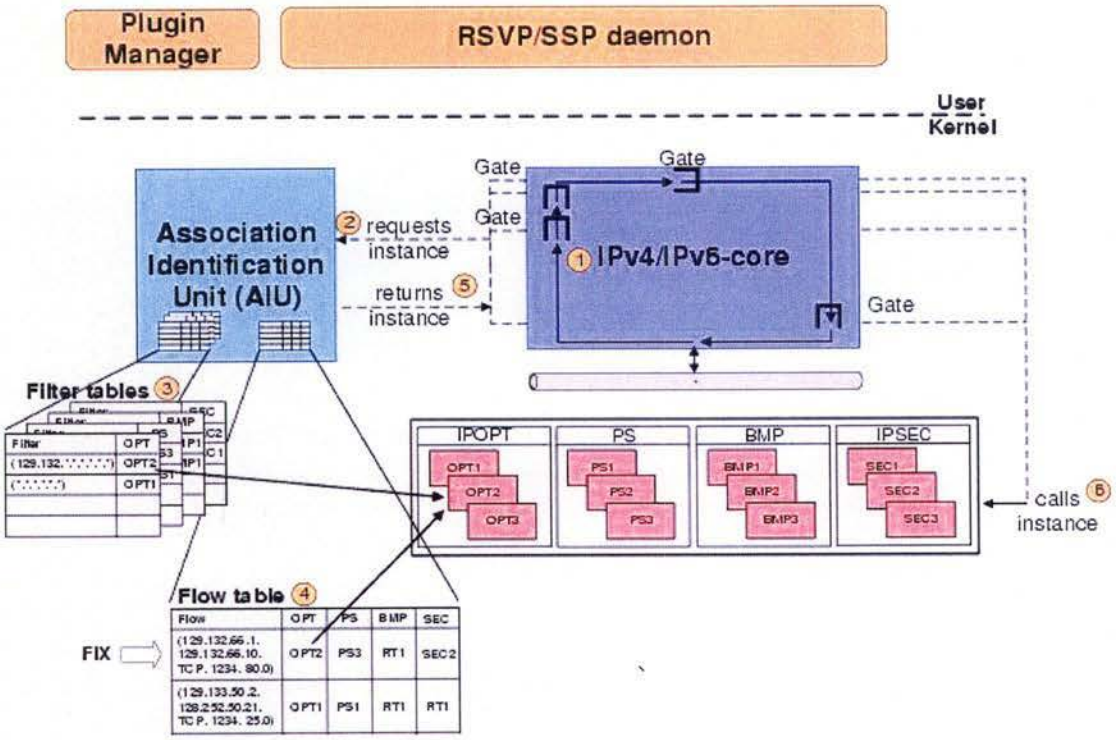


FIG. 1.10 – Architecture du système et chemin des données

une recherche dans la table des flux et constate qu'aucune entrée ne correspond à ce paquet. En conséquence, elle consulte alors la table des filtres correspondant à la porte des options IPv6. Le pointeur vers l'instance de plugin qui en résulte (OPT2 sur le schéma) sera renvoyé à la porte appelante.

- **4. Placement en cache du pointeur vers l'instance** : Avant que l'*AIU* ne retourne le pointeur à la porte, il stocke ce dernier dans la table des flux, avec comme entrée le six-tuple qui spécifie le filtre correspondant. Le paquet en question est probablement le premier d'une série. Le traitement des paquets du même flux qui suivront donnera donc lieu à une recherche concluante dans la table des flux. Ce mécanisme est notamment à la base des bonnes performances du système.
- **5. Envoi du pointeur vers l'instance** : Le pointeur vers l'instance du plugin trouvée est envoyé à la porte appelante.
- **6. Appel de l'instance** : La porte appelle l'instance du plugin en question, en passant la paquet en argument.
- **7. Répétition du cycle** : Au retour de l'appel à l'instance du plugin, le noyau IP continue de traiter le paquet. Au cas où celui-ci devrait rencontrer une autre porte, le cycle expliqué ci-dessus se répète de la même manière.

Achevons la présentation de cette architecture en précisant les quelques contraintes que les plugins doivent respecter afin de communiquer correctement avec le *PCU*.

Les plugins et le PCU

Comme nous l'avons mentionné auparavant, les plugins doivent satisfaire deux exigences. Premièrement ils doivent enregistrer une fonction de rappel (*callback function*) auprès du *PCU* quand ils sont chargés dans le noyau (*kernel*) du système. Ensuite, il faut que cette fonction de rappel puisse répondre à un ensemble de messages. Ces messages peuvent être regroupés en deux catégories : les messages standardisés et les messages spécifiques aux plugins. Examinons les messages standardisés qui requièrent une attention particulière :

- **create_instance** : Ce message permet de créer une instance d'un plugin. Il en résulte l'allocation d'une structure de données qui sera utilisée notamment pour stocker la configuration particulière du plugin et des informations sur l'état de l'exécution du système au moment de la création de l'instance. Une fonction de traitement effectif d'un paquet doit également y être spécifiée. Remarquons encore que la spécification d'autres fonctions optionnelles peut s'y trouver.
- **free_instance** : Il s'agit du message permettant de libérer une instance spécifique d'un plugin. Une instance ainsi libérée ne peut plus être utilisée par le noyau (*kernel*), et toutes les références vers celle-ci sont enlevées de la table des flux et de la table des filtres.
- **register_instance** : Cela permet d'enregistrer une instance de plugin auprès du *AIU* et de lier cette instance à un filtre qui doit être fourni en paramètre. Une même instance peut être enregistrée plusieurs fois, mais avec des spécifications différentes de filtres. Ce message se résume en fait à un appel d'une fonction d'enregistrement publiée par le composant *AIU*.
- **deregister_instance** : Ce message a pour conséquence d'enlever le lien entre un filtre spécifié dans le *AIU* et l'instance de plugin.

La création et la libération d'instances sont des tâches qui sont très particulières aux plugins. L'enregistrement et la désinscription d'un lien avec un filtre dans le *AIU* est une tâche relativement simple. En effet, dans la plupart des cas, cela se limite à un appel de fonction correspondante dans le *AIU*.

Le *PCU* lui même est d'ailleurs un composant très simple (200 lignes de code C) qui gère une table pour chaque type de plugin en stockant les fonctions de rappel et les noms de plugins. Une fois chargés dans le noyau, les plugins enregistrent leur fonction de rappel, via un appel de fonction au *PCU*. Tous les messages qui constituent l'information de contrôle en direction des plugins passent à travers le *PCU*. Généralement, de tels messages proviennent de l'espace de l'utilisateur (du *Plugin Manager* ou des démons). Le *PCU* est donc responsable de l'expédition de ces messages vers les plugins concernés, et du traitement de certaines exceptions.

1.5 Design Pattern

Comme prévu, nous allons ici présenter une analyse théorique particulière du principe du plugin, en fournissant un modèle de conception permettant de mettre en oeuvre ce même principe. Le type de modèle qui suit est appelé "*Design Pattern*" dans la littérature. Gardons à l'idée qu'il s'agit de modèles exclusivement conçus pour la programmation orientée-objet. La référence principale en terme de *Design Pattern* ([GHJV95]) nous dit que "*Chaque pattern décrit un problème qui intervient régulièrement dans l'environnement (du programmeur orienté-objet). Il décrit alors le squelette de la solution à ce problème, de telle façon que cette solution puisse être ré-utilisée un million de fois, sans faire exactement les choses deux fois de la même façon*".

Dans l'article définissant le pattern qui nous intéresse ([MMS02]), la justification de la nécessité d'une telle étude est basée sur le fait que les *design patterns* offrent les solutions les plus pratiques à tous les problèmes classiques de conception. De plus, on peut en tirer deux autres avantages principaux. D'une part, l'application de tels modèles peut fortement améliorer le *design* d'applications conçues par des programmeurs peu expérimentés. D'autre part, les architectures de logiciels basées sur les *patterns* sont plus facilement compréhensibles. Un grand nombre de *design patterns* ont déjà été définis et correspondent à la plupart des besoins. Cependant il manque encore un modèle adéquat et détaillé pour le concept de plugin, de plus en plus utilisé.

La pattern est présenté suivant le canevas proposé dans la référence [GHJV95, page 6].

1.5.1 Le pattern "Plugin"

Intention

Ce *pattern* explique comment concevoir une application de sorte qu'elle supporte le concept de plugin. Celui-ci permet à cette application d'être étendue pendant son exécution, en chargeant dynamiquement des modules ou des classes inconnues durant la compilation (de l'application) [20].

Motivation

Si, par exemple, une application capable d'afficher une variété de différents formats graphiques est créée, il se peut que le développeur ne soit pas capable dans un premier temps d'écrire les décodeurs correspondant à tous ces formats. Ce problème peut être résolu en permettant à des développeurs tiers d'implémenter des plugins qui reçoivent un flux de données contenant l'information codée et qui retournent une version décodée de ce flux. Un tel plugin

[20] Ceci confirme que nous sommes bien dans la définition donnée dans la section "Terminologie"

n'est chargé que si un format graphique approprié est demandé. Ceci permet à l'application de démarrer rapidement, puisqu'au début, aucun décodeur n'est chargé. Parallèlement à cela, la quantité de mémoire requise est fortement réduite.

Applicabilité

Ce *pattern* peut être utilisé pour répondre à de telles exigences :

- Besoins d'expansion pendant l'exécution, éventuellement inconnue au moment de la mise en route de l'application
- Modularisation de très grands systèmes, pour réduire la complexité
- Développement indépendant de composants du système, sans modifier d'autres modules et sans reconstruire le système entier
- Permettre le développement par des tiers, seulement sur base de la connaissance des interfaces
- Permettre le déploiement facile de nouveaux dispositifs et de nouvelles mises à jour, après la construction du programme
- Proposer des exigences faibles en termes de temps de démarrage et de hardware, particulièrement en ce qui concerne la mémoire
- Créer plus de flexibilité pour des serveurs en fonctionnement continu ne pouvant être relancés

Structure

Les classes et les interfaces du *pattern* sont illustrées par la figure 1.11, sous forme dun diagramme des classes au format UML 1.4.

Participants

PlugInLoader
- Recherche des implémentations de l'interface `PlugIn` pendant l'exécution, quand cela est nécessaire.
- Demande éventuellement à chaque implémentation concrète de plugin si elle doit être invoquée.
- Permet aux clients d'accéder aux plugins chargés (via un appel à la méthode `getPlugIns()`).

PlugIn (Interface)
- Fournit l'interface pour la communication avec toutes les implémentations concrètes de plugins du même type.
- Contient des méthodes (telles que `getName()`) pour accéder au nom du plugin, qui peut par exemple apparaître dans un menu.

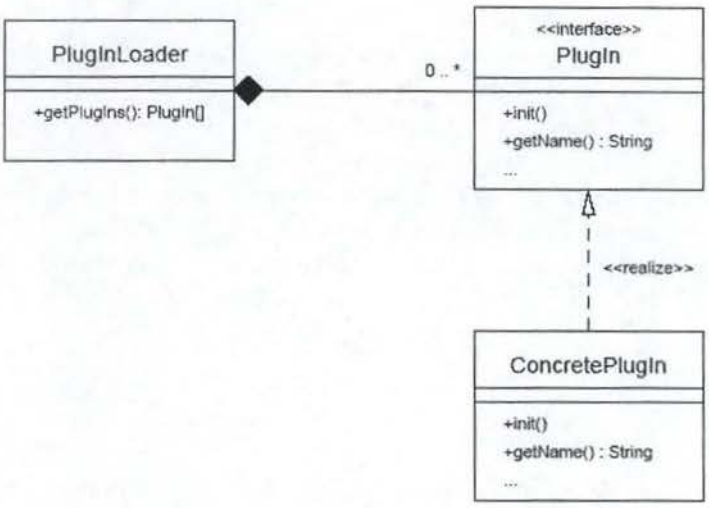


FIG. 1.11 – Diagramme des classes UML pour le *pattern* Plugin

- Contient éventuellement des méthodes, appelées méthodes de vote (*voting methods*), qui permettent aux plugins de voter pour ou contre leur invocation dans un contexte particulier.

Implémentation concrète de plugin (*ConcretePlugIn*)
- Implémente l'interface `PlugIn` et fournit des fonctionnalités spécifiques. Pour atteindre cet objectif, des modules indépendants de l'application (à côté des classes ajoutées par les programmeurs de plugins) peuvent être utilisés.
- Hérite éventuellement d'autres classes et peut également implémenter d'autres interfaces.

Collaborations

- Le `PlugInLoader` détermine le nom des plugins via un appel à la méthode `getName()` et leur permet de s'initialiser via la méthode `init()`.
- Un client du `PlugInLoader` peut accéder aux plugins en utilisant des méthodes de leur interface et le chargeur de plugin (PlugInLoader).
- Les implémentations concrètes de plugins (`ConcretePlugIn`) utilisent éventuellement des interfaces ou des classes fournies par une librairie de l'application.

Implémentation

Il est intéressant de donner quelques conseils techniques pour la recherche dynamique de sous-classes d'une interface nécessaire à l'implémentation du *pattern* Plugin. Il n'y a en réalité aucun langage de programmation capable de répondre directement à ce besoin. Par conséquent, pour déterminer des classes qui implémentent une interface donnée pendant l'exécution, un système de recherche de fichier peut être utilisé. En effet, dans les langages orientés-objet, il existe très souvent une correspondance directe entre une classe et un fichier. Cependant, cela sous-entend que le langage de programmation utilisé soit capable en cours d'exécution de charger des classes dont le nom n'est pas connu au moment de la compilation de l'application principale.

Echantillons de code

Deux morceaux de code d'applications utilisant le *pattern* plugin sont donnés en exemple. L'une est programmée en Java et l'autre en Perl. Ceux-ci sont fournis en Annexe 8 (page 111).

Expliquons cependant les principes sous-jacents à l'implémentation en Java. La technique, entièrement décrite dans [LB01], est basée sur la recherche dans un système de fichiers. Elle cherche en réalité toutes les sous-classes d'une interface ou d'une classe donnée. Le programme scanne des répertoires ou des fichiers d'archives (comme des fichiers jar par exemple). Le nom de la classe ou de l'interface, tout comme le nom du package, doit être spécifié. Une liste de toutes les classes qui sont des sous-types de la classe ou de l'interface donnée est alors renvoyée. Cette technique permet d'éviter également les conflits de type puisqu'un essai de chargement de chaque classe est réalisé, ainsi qu'un test de type. Pour remplir cet objectif, on utilise la méthode `Class.forName()` qui crée un objet de type `Class` du nom de la classe en question. Il est également intéressant qu'une classe qui implémente une interface de plugin possède un constructeur public sans paramètre, de façon à permettre l'instantiation. Ceci se retrouve régulièrement dans le contexte des "*Java Beans*" [21] ([HC00, chapitre 8]), et aucune restriction n'est à relever. Il est alors possible d'utiliser la méthode `newInstance()` sur l'objet `Class` pour construire un nouvel objet.

Utilisations connues

Le programme de traitement d'image Gimp[22] permet également d'ajouter de nouvelles fonctionnalités grâce à l'utilisation de plugins. On peut également mentionner le "SLC por-

[21] "Java Beans" est une technologie Java permettant de construire des composants, c'est-à-dire des briques applicatives réutilisables. Les Java Beans peuvent être interrogés sur leurs fonctionnalités et sont manipulables graphiquement. Ils interagissent de façon dynamique entre eux et avec d'autres composants et constituent une technologie propice à la mise en oeuvre d'une architecture basée sur des plugins.

[22] http ://www.gimp.org

tal"[23] et l'interface graphique de "GeoStoch"[24], qui utilisent également ce concept.

Patterns en relation

Les explications qui suivent ont pour objectif de présenter des similitudes entre le *pattern* que nous venons de décrire et d'autres faisant partie de l'existant. N'étant pas indispensables à la compréhension du *pattern* plugin, elles s'adressent principalement au lecteur curieux ou au programmeur ayant une certaine expérience en matière de *design patterns*.

Signalons tout d'abord le *pattern "Pluggable Components"* ([Völ99]) qui est quelque peu similaire à celui décrit ici.

Le *pattern "Command"* ([GHJV95]) est assez semblable aux plugins qui exécutent une commande. La principale différence réside dans le fait que les plugins ne sont pas connus au moment de la compilation. C'est pour cette raison que leur nom ne peut donc pas se trouver dans le code source de l'application.

Un plugin concret, c'est-à-dire une implémentation de l'interface `PlugIn`, est généralement basé sur un certain nombre de classes spécialement utilisées à cet effet. C'est en réalité une application du *pattern "Facade"* ([ST02]), dans laquelle la classe `ConcretePlugIn` joue le rôle de la façade. En outre, on peut également dire que l'interface et le chargeur des plugins constituent la façade derrière laquelle les plugins concrets sont cachés.

Par ailleurs, la méthode `getPlugIns()` du `PlugInLoader` est similaire à une méthode de type *factory*. Les principales différences résident dans le fait que `getPlugIns()` peut retourner plusieurs instances et que les classes sont cherchées pendant l'exécution du programme par le chargeur de plugins, alors que les noms des classes sont déjà codés dans l'implémentation d'une méthode *factory* (voir *pattern "Factory Method"* dans [GHJV95]).

Enfin, il peut parfois être souhaitable que des plugins d'un certain type soient chargés au maximum une fois. Dans ce cas, afin de garantir cette propriété, le `PlugInLoader` doit simplement implémenter le *pattern "Singleton"* ([GHJV95]).

1.5.2 Décomposition en couches

Les auteurs du *pattern* "plugin" ont également proposé une généralisation des concepts présents dans ce dernier. Le résultat de cette généralisation prend la forme d'une structure en couches d'une application basée sur les plugins. La figure 1.12 illustre cette découpe.

Dans cette structure, il peut y avoir plusieurs interfaces de plugins qui cohabitent. Néan-

[23] https ://slc.mathematik.uni-ulm.de
[24] http ://www.geostoch.de

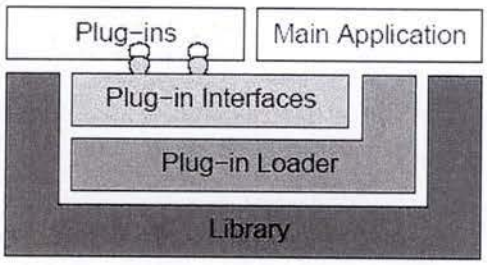


FIG. 1.12 – Couches d'une application basée sur les plugins

moins, un "plugin concret" implémente une et une seule de ces interfaces. En plus du code des plugins, les classes de la librairie sont utilisées pour remplir des tâches communes à plusieurs plugins. La librairie contient des routines générales qui, si elles ne se trouvaient pas là, auraient du être implémentées par un certain nombre de plugins ou par l'application principale. Toute fonctionnalité supplémentaire est fournie par les plugins, qui peuvent appartenir à différentes catégories , en fonction de l'interface qu'ils implémentent. Pendant l'exécution du programme principal, les plugins sont chargés par le chargeur de plugins (*Plug-in Loader*), éventuellement après que leur utilisation ait été requise. Le chargeur gère également tout ce qui à trait aux plugins. Typiquement, ces derniers sont initialisés et leur nom est recherché. Quand l'application principale ou une librairie veut accéder à un certain plugin, le chargeur est interpellé. L'application principale (*Main Application*) est implémentée "au dessus" de la librairie et peut accéder à tous les plugins, à travers le chargeur, et via les interfaces. Toute autre partie du système qui ne peut pas être modélisée comme un plugin fait partie de l'application principale.

Chapitre 2

Développements théoriques

2.1 Objectifs

Le but de ce chapitre est d'arriver, à la lumière des informations présentées dans le chapitre précédent, à des conclusions théoriques concernant le concept de plugin. Il faudra pour commencer rassembler les fonctions que doit remplir un système qui met en oeuvre le concept qui nous intéresse. Nous devrons ensuite proposer une architecture de composants remplissant ces fonctions, suffisamment générique pour pouvoir être appliquée à la plupart des applications basées sur les plugins. Nous regarderons alors si cette architecture peut s'appliquer aux cas présentés dans le premier chapitre.

2.2 Méthodologie

Afin d'aboutir à des conclusions applicables à tous types de programmation, nous allons devoir raisonner à un niveau d'abstraction relativement élevé. A un tel niveau, il n'existe pas véritablement de technique bien définie permettant d'inférer des résultats pertinents. Par conséquent nous ne suivrons pas une méthodologie précise pour arriver à nos fins.

Nous nous contenterons, dans un premier temps, de retracer les étapes nécessaires au bon fonctionnement d'une application qui fait usage de plugins, afin de mettre en évidence les fonctions qui doivent être remplies par le système.

Par après, nous essayerons de définir un certain nombre de composants qui remplissent chacun une ou plusieurs des fonctions précitées. Remarquons déjà que l'architecture ainsi construite devra être suffisamment générale pour être utilisée par différents paradigmes de programmation, mais aussi assez précise et complète pour intégrer tous les concepts sous-jacents au principe du plugin. Nous regarderons brièvement comment l'information concernant la gestion des plugins est véhiculée entre les composants. On pourra enfin considérer que l'application de l'architecture proposée aux cas du premier chapitre constituera une validation du modèle.

Nous regrouperons enfin les détails d'implémentation dont nous disposons pour mettre en évidence les instructions-clés permettant d'implémenter un système de plugins en Java.

2.3 Fonctionnement d'une application basée sur les plugins

Pour nous aider à déceler les différentes tâches que doit accomplir un système utilisant le principe du plugin, essayons de nous imaginer le fonctionnement d'un tel programme. Nous ferons cela intuitivement, et au vu de ce qui a été présenté auparavant.

Supposons donc que le programme en question est en cours d'exécution. Nous avons vu

dans le chapitre précédent que la requête d'utilisation d'un plugin pouvait émaner de l'utilisateur mais également de l'application elle-même. Dans ImageJ, par exemple, l'utilisateur du programme peut charger lui-même un plugin et presser sur l'item qui lance son exécution. Par contre, une fois les plugins chargés dans Nature Life's, c'est le système lui-même qui utilise les caractéristiques spécifiques des comportements et des espèces, afin de décider le cap que prendra une bête. On suppose également que l'on connaît des informations suffisantes pour trouver le plugin à utiliser. Dans certains cas il s'agit du nom précis du plugin (ImageJ). Dans d'autres cas, l'information peut se résumer en la catégorie à laquelle le plugin appartient. Rappelons nous, par exemple, qu'un paquet qui traverse le système Router Plugins est simplement dirigé vers une porte correspondant au type de plugin à lui appliquer. Cela implique la présence d'un mécanisme capable de retrouver l'instance précise du plugin à utiliser. C'est bien le cas dans Router Plugins.

Deux possibilités se présentent alors par la suite. Soit le plugin est déjà connu de l'application et a déjà été chargé, soit il n'a pas encore été lié au noyau du programme principal et cela reste à faire. Dans ce second cas, il faut le charger de façon à ce qu'il puisse être utilisé correctement par le système. Cette action de chargement est comparable à un enregistrement du plugin auprès du programme principal. Elle implique que la partie de code qui constitue le plugin et qui était inconnue auparavant, puisse être rendue accessible au système pour exécution. Restons générique et admettons que le chargement dont il est question correspond le plus souvent à l'allocation d'un espace mémoire réservé au code du plugin. La partie du système chargée de l'exécution de ce dernier devra donc pouvoir retrouver cette zone pour "passer la main" au plugin. Nous supposons à présent que cette étape s'est déroulée avec succès.

Il reste donc maintenant à exécuter le plugin. Comme celui-ci était inconnu avant son chargement, cela suppose que l'on connaisse un moyen de communiquer avec lui. Différents cas de figure peuvent à nouveau se présenter. En effet certains plugins doivent simplement être "lancés" afin que la main (comprenons le processeur) leur soit cédée. Le code du plugin est alors exécuté à partir du début. On considère que l'exécution se termine quand toutes les opérations précisées dans la définition du plugin ont été achevées. C'est le cas par exemple du plugin "Inverter_" d'ImageJ, présenté en Annexe 6 (page 102). Effectivement, une fois déclenché, ce plugin inverse les couleurs de l'image qui se situe dans la fenêtre courante de l'application, et son action se termine sans suite. Dans ce cas, il faudra donc connaître la commande permettant de démarrer son exécution. Néanmoins, il se peut aussi que la communication entre le système et le plugin soit continue et ne se limite pas à une commande de lancement. Le système est alors conscient de la présence du plugin (qui a été chargé) et il s'adresse à lui quand cela est nécessaire. C'est précisément ce qu'il se passe dans Nature Life's lorsque l'application invoque notamment les méthodes `drawYou` et `getAbout` des plugins de type `Beast`, ou les méthodes `giveCourses` et `getAbout` des plugins de type `Behaviour` (voir pages 12 et 13). Il est clair que, dans ce cas, il faut que le programme principal connaisse le moyen de s'adresser au plugin pour lui demander d'effectuer certaines tâches particulières. Remarquons que Router Plugins se situe à mi-chemin entre les deux catégories expliquées ci-dessus. En effet, lorsque le *AIU* informe une porte de l'instance de plugin à utiliser, la fonction

de traitement du paquet implémentée par cette instance est simplement invoquée (voir page 33). Ceci correspond au premier cas. Cependant, n'oublions pas qu'il existe un ensemble de messages adressés par le *PCU* aux plugins, auxquels ces derniers doivent pouvoir répondre pour assurer le bon fonctionnement du système (voir page 34). Nous nous rapprochons alors très fort de la seconde catégorie. En conséquence, nous constatons que, dans le premier comme dans le second cas, il faut qu'il y ait eu préalablement définition d'un langage permettant la communication entre le noyau de l'application et les plugins.

Enfin, une fois le plugin utilisé, il est également intéressant de savoir si celui-ci servira encore ou pas. En cas de réponse négative, il est préférable que les ressources utilisées soient libérées. Cela laisse sous-entendre la nécessité d'un mécanisme qui gère le déchargement des plugins.

2.4 Proposition d'architecture

A la lumière de ce qui vient d'être dit dans la section précédente, nous pouvons à présent tenter de dessiner une architecture-type reprenant les composants et la communication au sein d'une application basée sur les plugins. Remarquons déjà que la structure en couches présentée à la fin du chapitre précédent semble être une bonne base à raffiner quelque peu. La figure 2.1 illustre cette architecture.

Examinons les différents membres de cette architecture :

Les composants tiers

Nous considérerons qu'il s'agit, en dehors de l'interface (graphique) et des composants spécialement créés pour gérer le système des plugins, de l'ensemble des autres composants de l'application principale. C'est en réalité un genre de fourre-tout dans lequel nous rangeons tous les éléments de l'architecture du logiciel qui n'intéressent pas directement notre analyse.

Le (G)UI

Il s'agit de l'interface, éventuellement graphique, qui permet à l'utilisateur d'interagir avec le logiciel. Il parait bon d'en faire un composant de notre architecture, étant donné qu'elle constitue un point commun à la plupart des logiciels modernes. De plus, un bon nombre de requêtes d'utilisation de plugins proviennent de celle-ci. Nous considérerons également que les résultats graphiques de l'exécution des plugins sont affichés à l'aide de ce composant.

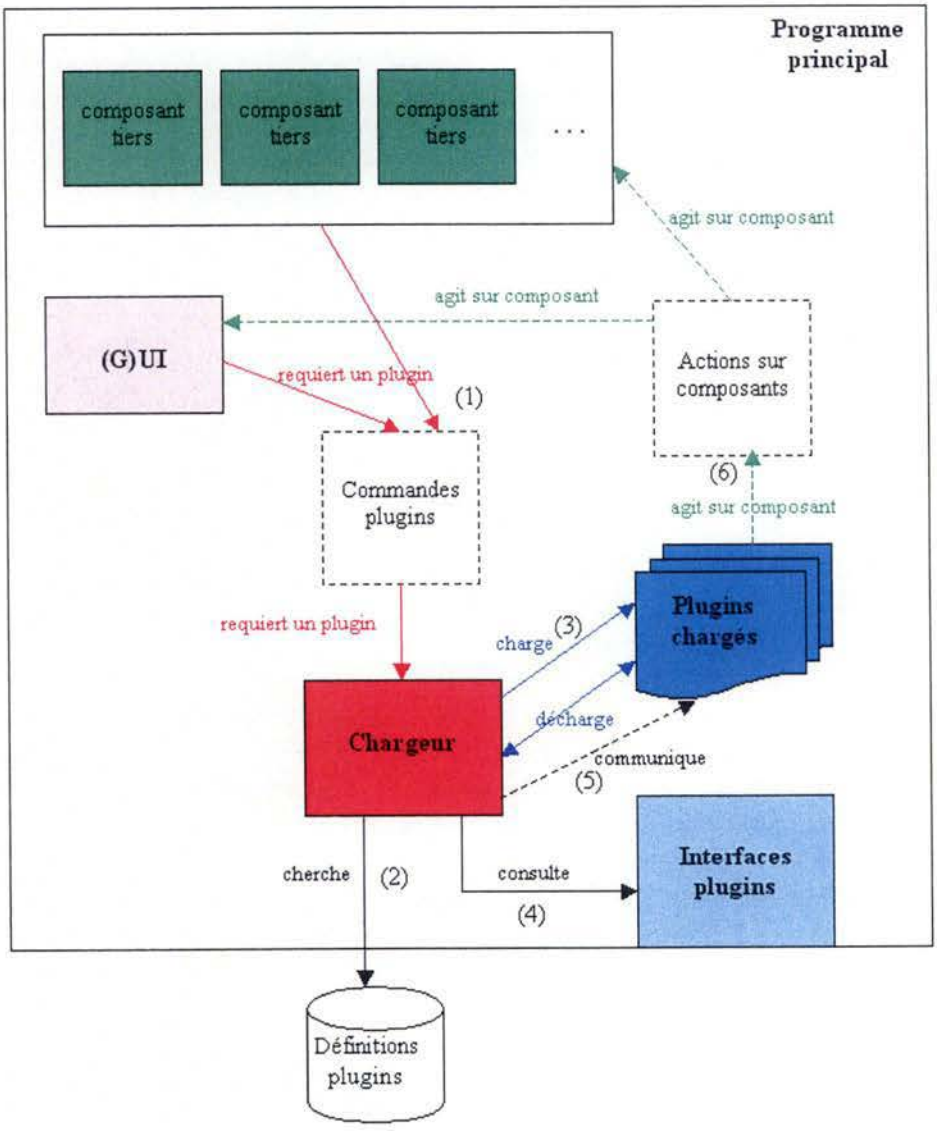


FIG. 2.1 – Architecture générique d'une application basée sur les plugins

Le chargeur (de plugins)

Le chargeur est bien entendu le composant central dans la gestion des plugins. En effet, il est responsable des différentes tâches qui permettent à notre système de fonctionner correctement.

Premièrement, il doit être capable de retrouver la bonne instance du plugin à utiliser, suite à une demande d'utilisation de la part des composants tiers ou du GUI. Rappelons qu'il reçoit de ceux-ci des informations parfois incomplètes, telles que le type de plugin à utiliser. Il abrite donc un mécanisme permettant de retrouver le plugin sur base de ces informations.

Ensuite, le chargeur doit pouvoir analyser si le plugin à utiliser est déjà chargé ou pas. S'il ne l'est pas, c'est lui qui est responsable de l'action de chargement que nous avons décrite dans la section précédente. Signalons simplement que le chargement se fait en général suite à une recherche dans l'espace des définitions des plugins. Cet espace se situe en dehors du noyau de l'application, sur une zone mémoire dont le chargeur doit connaître l'emplacement.

Puisque le chargeur est responsable de la gestion des instances de plugin, il ne parait pas essentiel de considérer l'existence d'un autre composant responsable de leur accomplissement. En effet, à partir du moment où il gère les emplacements des différents plugins, il n'y a qu'un pas à réaliser pour lancer leur exécution et communiquer avec eux quand cela est nécessaire. Ce pas est d'ailleurs franchi à partir du moment où la chargeur connaît les commandes d'exécution et de communication auxquelles les plugins peuvent répondre. Ce problème est résolu grâce à la consultation des interfaces des plugins, dont nous parlerons au point suivant. Il y a cependant une remarque qui se doit d'être faite à propos de l'exécution des plugins. Nous avons auparavant émis l'hypothèse que le chargeur était responsable du lancement des plugins ou de la communication avec ces derniers. Or, il se peut très bien que ces actions soient effectuées directement par des autres composants du système. Néanmoins, pour que cela puisse se faire, il faut que ces composants aient reçu au préalable des informations sur l'emplacement des plugins (des pointeurs, en terme informatique). Ces informations ne peuvent être fournies que par le chargeur qui gère ces emplacements. Pour cette raison, nous avons simplifié le schéma en considérant que toute communication avec les plugins provient uniquement du chargeur. Notons encore que sur le schéma, le démarrage de l'exécution des plugins et la communication avec ces derniers sont regroupés sous le libellé "communique".

Enfin, une autre responsabilité du chargeur peut être le déchargement d'un plugin. Cela correspond surtout à la libération des ressources utilisées par ce dernier. Remarquons que la flèche représentant le déchargement a volontairement été dessinée à double sens. On a voulu par là exprimer le fait que le désir de décharger un plugin ne vient pas forcément du chargeur. Certes, nous avons par exemple vu dans Router Plugins qu'il existe un composant qui envoie un message à une instance de plugin lorsque celle-ci doit se libérer. Dans ce cas la flèche va du chargeur vers les plugins. Cependant, il ne faut pas négliger le cas des plugins, qui, une fois lancés, s'exécutent de façon autonome jusqu'à ce que leur tâche soit achevée. Ceux-ci peuvent alors très bien être libérés automatiquement, auquel cas le chargeur doit en être informé. Il est

cette fois plus logique que la flèche parte des plugins vers le chargeur. On peut exprimer ces principes en termes de fournisseur et d'utilisateur de service. Le service en question, fourni par le chargeur, correspond au déchargement du plugin. Dans le premier des deux cas expliqués ci-dessus, on considère que le chargeur a lui-même recours à son propre service, lorsque ses mécanismes internes de gestion de plugins en constatent la nécessité. Pour le deuxième cas, les plugins dont l'exécution vient de se terminer deviennent les utilisateurs du service offert par le chargeur, dans le sens où ils informent ce dernier de l'utilité de la libération des ressources.

Les interfaces (des plugins)

Les interfaces définissent le langage de communication avec les différents types de plugins. De manière générale, nous dirons qu'elle regroupent des ensembles de messages auxquels les plugins doivent pouvoir répondre afin d'être valides. Insistons sur le caractère très abstrait de ce composant, tant celui-ci peut être mis en oeuvre de façons différentes en fonction du type et du langage de programmation utilisés.

Remarquons également que ce composant a volontairement été placé à la frontière entre le noyau de l'application et l'extérieur de celui-ci. Nous voulons ici traduire le fait qu'il s'agit normalement de la seule partie de code qu'il est indispensable au programmeur tiers de connaître pour que son plugin soient acceptable[1].

Les plugins chargés

Il s'agit des plugins à proprement parlé, dont tout ou une partie du code s'exécute en fonction de la catégorie dont il s'agit (voir section précédente). Ce ne sont pas des composants persistant dans le système puisqu'il sont chargés et déchargés en fonction des besoins du programme et de l'utilisateur. Tous ces plugins sont capables de répondre aux messages définis par l'interface à laquelle ils correspondent. Notons également qu'ils sont capables d'agir sur d'autre composants du système, telle l'interface graphique. C'est notamment le cas du plugin "Inverter_" d'ImageJ que nous avons déjà évoqué et dont le code se situe en Annexe 6 (page 102).

Commandes plugins

Cet élément est en réalité un composant fictif ajouté pour simplifier l'architecture. De plus, sa présence laisse sous-entendre un traitement particulier des requêtes d'utilisation des plugins. Etant donné que les commandes en direction des plugins peuvent émaner des composants tiers ou de l'utilisateur à travers la GUI, on considère qu'il est responsable de la gestion de ces

[1] Nous parlons évidemment du code du système de gestion en lui-même. Il est clair que le programmeur doit également connaître le moyen d'interagir avec d'autres composants du système pour que son plugin remplisse correctement son rôle.

requêtes et de leur propagation vers le chargeur. Par ailleurs, il est clair que les messages dont il est ici question correspondent à ceux prévus par les interfaces des plugins. Ajoutons enfin que l'on peut voir ce composant fictif comme un service qui peut être rendu par le chargeur, c'est-à-dire le traitement des commandes définies dans les interfaces.

Actions sur composants

Il s'agit également d'un composant fictif créé pour les mêmes raisons que le précédent. Comme nous l'avons déjà souligné, les plugins peuvent agir sur l'ensemble des composants tiers et de la GUI. Le rôle de ce composant est de regrouper l'ensemble de ces actions sur le reste du système.

2.5 Parcours de l'information

L'architecture proposée permet de respecter les étapes commentées dans la section 2.3. Leur chronologie est d'ailleurs représentée sur la figure 2.1 à l'aide de numéros de 1 à 6. Parcourons-les brièvement en examinant comment l'information est véhiculée entre les composants :

1. Une action lancée par un des composants tiers ou l'utilisateur (via la GUI) requiert l'utilisation d'un plugin. Les informations permettant de déterminer l'instance à utiliser sont acheminées au chargeur.
2. Supposons que l'instance à utiliser n'est pas encore connue du système. Le chargeur cherche alors la définition du plugin qui l'intéresse dans ce que nous avons appelé l'espace des définitions.
3. Une fois la définition du plugin récupérée, le chargeur alloue les ressources nécessaires à l'exécution du plugin. Selon le vocabulaire utilisé jusqu'à présent, nous dirons qu'il charge le plugin.
4. Il reste alors à exécuter le plugin. Pour savoir comment s'adresser à ce dernier, le chargeur doit se servir des informations définies dans l'interface correspondant au plugin en question.
5. Le chargeur connaît à présent le vocabulaire utilisé par le plugin. Il peut alors lancer l'exécution de celui-ci ou bien commencer la communication (en fonction des catégories de plugins établies précédemment).
6. L'action du plugin peut alors se faire ressentir sur différents composants de l'application.

Par après il faudra bien sûr que le système libère les ressources utilisées par le plugin. Mais comme nous l'avons déjà laissé sous-entendre, cela ne se fait pas nécessairement de manière automatique après chaque exécution d'un plugin.

Il reste une remarque à faire à propos de l'architecture dessinée. Le schéma comprend, pour représenter l'information qui circule, des flèches pleines et des flèches en pointillés. Ces dernières regroupent l'information qui est directement en relation avec l'exécution d'un plugin. C'est en quelque sorte l'information que doit pouvoir coder un programmeur de plugins. Il est donc logique que ces flèches n'apparaissent qu'au moment où le chargeur entame la communication avec le plugin. En ce qui concerne les flèches pleines, nous pouvons considérer qu'elle représentent l'information directement relative à la gestion des plugins.

2.6 Validation de l'architecture

Le but de cette section est de voir si nous pouvons faire entrer la découpe des différents cas du premier chapitre dans l'architecture proposée précédemment. Nous entendons en réalité regrouper différents éléments des informations dont nous disposons, en composants comparables à ceux définis dans le modèle générique. Soyons conscients du fait que nous allons réaliser une démarche inverse par rapport aux étapes classiques de conception d'un logiciel. En effet, en génie logiciel, l'implémentation de l'application vient à la fin du processus alors que la conception de l'architecture a lieu au début. Soulignons également que nous disposons d'informations très différentes selon les cas. Pour les deux raisons qui viennent d'être citées, il est clair que nous allons devoir réaliser un certain nombre d'adaptations et d'abstractions sur les données possédées. Nous considérerons alors que parvenir à établir une similitude entre notre architecture générique et la structure des cas présentés constituera une validation de notre modèle.

Une première remarque qui doit être faite est que nous n'allons pas nous attarder à rechercher dans nos données les opérations correspondant aux libellés "requiert un plugin" et "agit sur composant" de notre architecture générique. En effet, il existe forcément de tels mécanismes dans les logiciels que nous analysons, et il ne s'agit pas là des opérations essentielles à la compréhension des mécanismes qui nous intéressent. Pour les mêmes raisons, nous ne chercherons pas à établir des correspondances entre des éléments des applications étudiées et les composants "Commandes plugins" et "Actions sur composants" du modèle générique.

2.6.1 Nature Life's

En partant de la découpe en packages donnée dans le premier chapitre, il est assez facile d'établir une correspondance entre la structure de Nature Life's et notre architecture générique. Cette correspondance est illustrée par la figure 2.2.

Le type d'abstraction à effectuer ici consiste simplement à considérer certains packages comme étant l'implémentation de certains composants. On remarquera d'ailleurs que l'entièreté des packages de Nature Life's peut être englobée dans l'architecture.

Les composants tiers correspondent en fait aux packages `area`, `intelligence`, et `obstacle`. Le composant "GUI" est lui comparable au package `natureGUI`. Au lancement de l'application, toutes les définitions de plugins situées dans le répertoire adéquat donnent lieu à un chargement, de sorte que chaque plugin puisse être instancié. C'est d'ailleurs le seul moment où les plugins sont chargés. Le désir de charger des plugins vient donc du programme lui-même (les composants tiers), mais les informations sur les plugins chargés sont récupérées pour compléter l'interface graphique. Les deux flèches rouges peuvent donc être conservées. De plus, l'application a besoin d'interroger les plugins tout au long de la simulation. Pour réaliser ces différentes opérations, on a recours au chargeur, qui correspond à la classe `PluginLoader` du package `plugins`.

Pour retrouver la définition des plugins qu'il doit charger, le `PluginLoader` utilise sa méthode `loadClasses`, qui cherche dans un répertoire déterminé des fichiers `".class"` correspondant à des plugins valides. Cette méthode détermine d'ailleurs la validité de ces définitions en fonction du fait qu'elles implémentent correctement ou non les bonnes interfaces. C'est ce que représente dans ce cas la flèche libellée "consulte". Les interfaces en question sont également très faciles à retrouver. En effet, elles correspondent parfaitement avec le package `pluginSDK`, qui contient l'interface `Plugin`, et les classes `Beast` et `Behaviour`. Nous ne reviendrons plus sur la relation entre ces trois entités java, expliquée de manière détaillée à partir du diagramme des classes de la page 15.

L'opération de chargement d'un plugin s'achève lorsque le `PluginLoader` en a créé une instance. Pour ce faire, il fait appel à sa méthode `createInstance`, qui construit un objet correspondant soit à une espèce, soit à un comportement. Le plugin est alors chargé complètement et est prêt à être exécuté. Rappelons que le code de la classe `PluginLoader` est fourni en Annexe 3 (page 93).

Dans Nature Life's les plugins sont l'objet d'une communication continue avec le reste du système. Celle-ci se fait grâce aux appels aux différentes méthodes qu'implémentent les plugins. Par exemple, à chaque fois que l'application doit effectuer un pas de la simulation, elle appelle la méthode `giveCourses` pour l'ensemble des comportements de chaque bête présente sur l'aire de jeu. Cela permettra de déterminer le cap que l'animal en question devra suivre. Les appels aux différentes méthodes des plugins sont illustrés par la flèche libellée "communique".

L'action des plugins se fait bien entendu ressentir sur les autres composants. Par exemple, la confrontation des différents caps possibles que l'animal peut prendre (déterminés avec les appels à `giveCourses`) donnera lieu à un mouvement de l'animal sur le terrain de jeu. Ceci justifie le maintien des flèches vertes.

Il reste enfin à traiter du déchargement des plugins. Il faut savoir qu'en Java, il existe un mécanisme appelé *Garbage Collector* qui "tue" les objets qui ne sont plus utilisés et libère donc les ressources qui leur correspondent. Par conséquent, lorsqu'une bête est tuée par un prédateur ou par l'utilisateur, les mécanismes Java sous-jacents libèrent les ressources

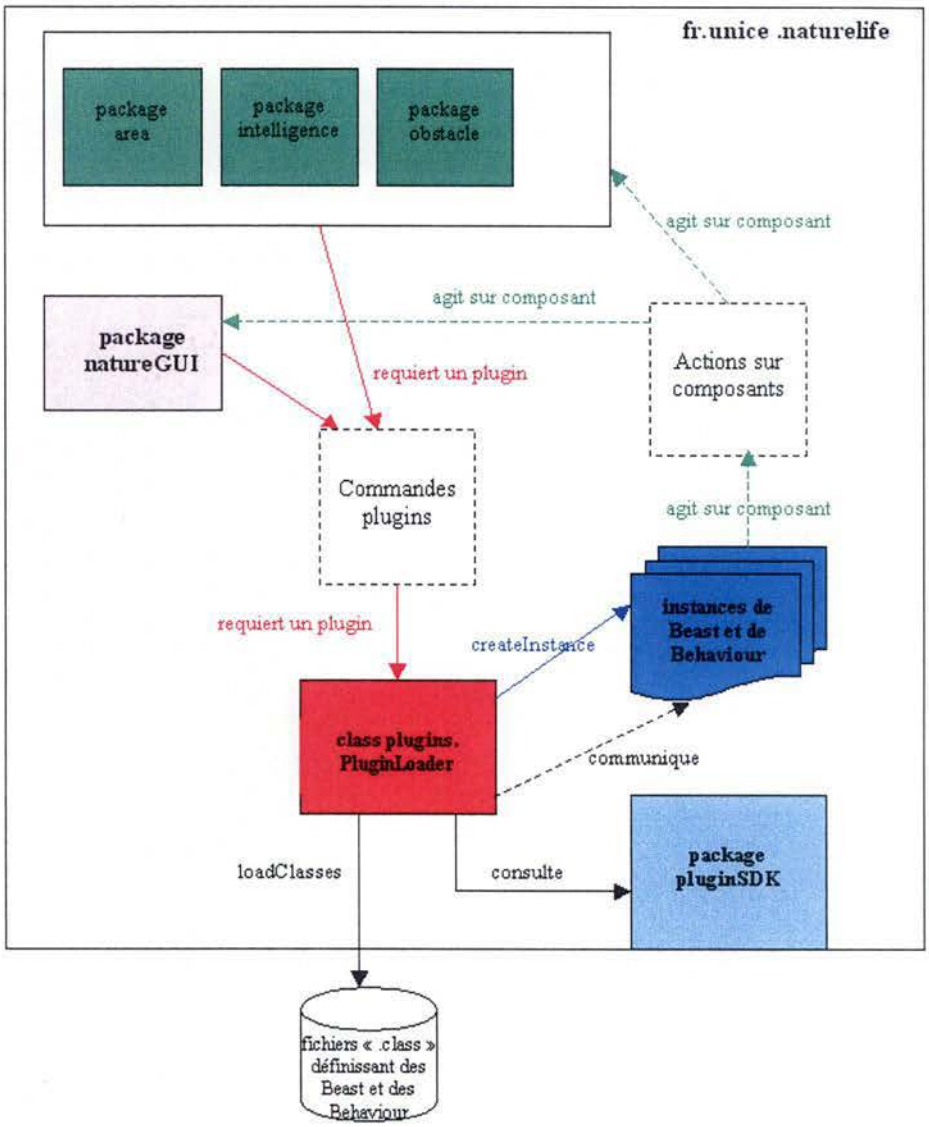


FIG. 2.2 – Correspondance entre l'architecture générique et la structure de Nature Life's

utilisées par celle-ci. De plus, les plugins de Nature Life's doivent rester chargés pendant toute l'exécution du logiciel, de façon à ce que de nouvelles bêtes puissent être ajoutées sur le terrain de jeu. Il n'y a donc pas lieu dans ce cas d'implémenter une opération de déchargement des plugins. Ceci justifie la disparition de la flèche libellée "décharge" dans notre architecture générique.

2.6.2 ImageJ

Comme pour Nature Life's, nous établirons la correspondance entre la structure d'ImageJ et le modèle générique à partir de la découpe en packages. Celle-ci est explicitée dans le premier chapitre à la page 24. Nous aurons par conséquent recours aux mêmes types d'abstractions. Nous devrons cependant morceler quelque peu certains packages, mais nous verrons que la similitude est assez directe. La figure 2.3 présente cette correspondance.

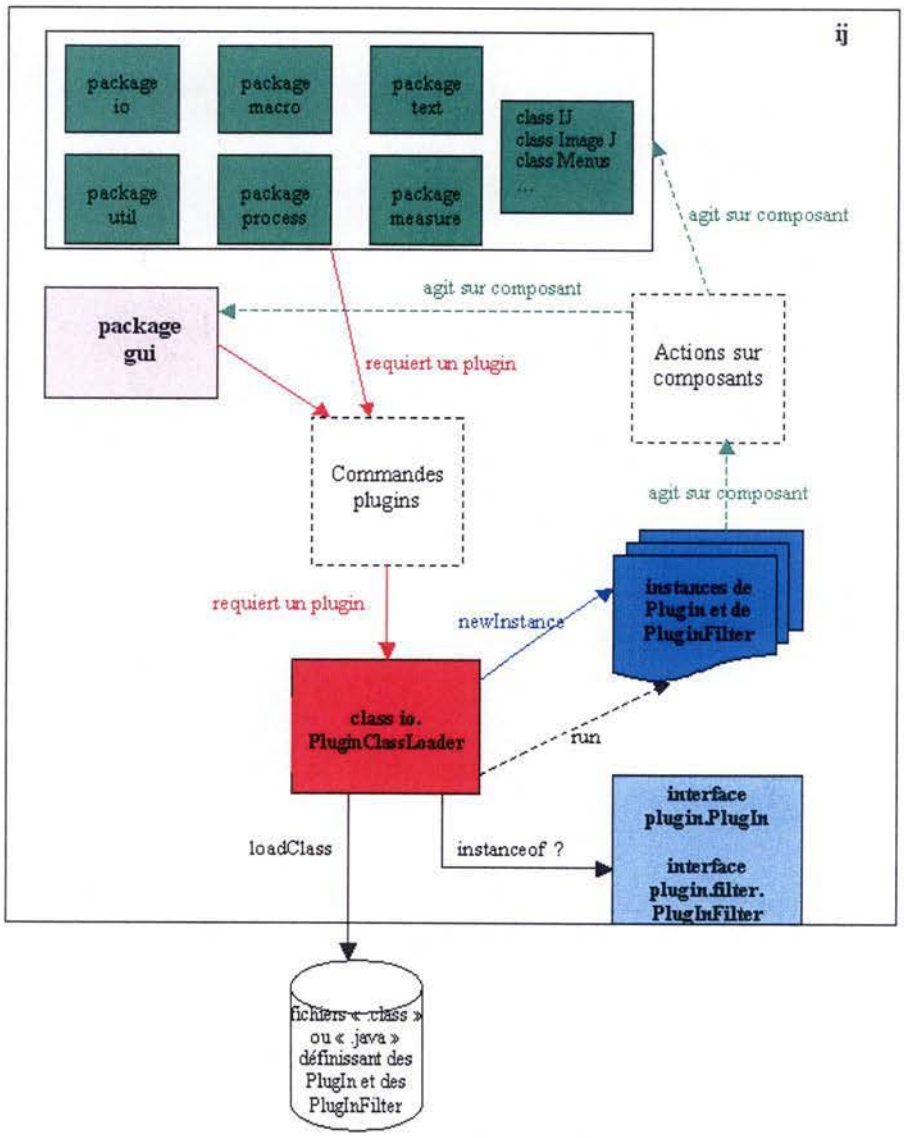


FIG. 2.3 – Correspondance entre l'architecture générique et la structure d'ImageJ

Les composants tiers peuvent être assimilés aux packages `io` (à l'exception de la classe

`PluginClassLoader`), `macro`, `text`, `util`, `process` et `measure`. On y ajoutera également les classes seules telles que `IJ`, `ImageJ`, `ImagePlus`, etc. Le GUI peut sans problème être confondu avec le package `gui` d'ImageJ.

Il y a plusieurs façons d'avoir recours à un plugin dans ImageJ. La demande peut venir de l'utilisateur, via l'interface graphique. Celui-ci peut soit lancer l'exécution d'un plugin déjà connu de l'application, soit charger un nouveau plugin que l'application aura au préalable pris soin de compiler. Nous avons vu précédemment que la plupart des outils d'ImageJ étaient implémentés sous la forme de plugin. Par conséquent, la demande d'utilisation peut également émaner de l'application elle-même (entendons les composants tiers), quand celle-ci doit utiliser un de ces outils. Les deux flèches rouges peuvent donc être conservées.

Dans ImageJ, le chargeur n'est pas implémenté par un package entier mais par la classe `io.PluginClassLoader`. Lorsque celui-ci doit charger un plugin, il a deux types de sources possibles. En effet, si le plugin n'est pas connu de l'application, l'utilisateur utilise la commande "compile and run" et le chargeur effectue une recherche dans un répertoire particulier, qui contient des fichiers `".java"`. Il compilera alors le fichier trouvé pour placer le fichier ".class" correspondant dans le package approprié. Si au contraire le plugin est déjà connu par l'application, un fichier `".class"` existe déjà et le chargeur consulte directement le package où se situe ce fichier. Il reste alors à charger concrètement le plugin à partir du fichier ".class". Pour ce faire, le chargeur utilise sa méthode `loadClass`. Notons que l'exécution d'un plugin est initiée par des méthodes situées dans la classe `IJ`. Cependant, ces méthodes font appel à une instance de `PluginClassLoader` pour effectuer le chargement. Nous restons donc cohérents avec notre schéma de base.

Il faut encore créer une instance du plugin en question, ce qui est réalisé grâce à l'instruction `newInstance`. Le chargeur doit également vérifier la validité du plugin instancié grâce aux interfaces (voir la flèche "consulte" dans le modèle). Cette opération est effectuée via l'instruction booléenne `instanceof`. Rappelons que les parties de code-source comprenant ces méthodes et ces instructions sont fournies en Annexe 7 (page 104).

Il reste alors à exécuter concrètement le plugin. Nous savons que les plugins d'ImageJ s'exécutent de manière autonome une fois lancés. Pour ce faire on utilise la méthode `run`, qui doit être définie par chaque plugin. Nous pouvons faire ici le même genre de remarque que pour l'architecture générique, dans le sens où cette commande n'est pas directement faite par le chargeur, mais par la classe `IJ`. Cependant, comme nous l'avons déjà dit, cette commande ne peut avoir lieu qu'après que le programme ait récupéré l'instance du plugin à exécuter. Cette instance étant fournie par le chargeur nous pouvons considérer qu'il n'est pas faux de faire partir la commande `run` du `PluginClassLoader`.

Le maintien des flèches vertes représentant l'action du plugin sur les composants du système est assez facile à justifier. Par exemple, pensons simplement au plugin "Inverter_" déjà évoqué auparavant (voir notamment Annexe 6, page 102). Celui-ci inverse les couleurs de l'image située dans la fenêtre courante. Dans ce cas, son action se fait directement ressentir

sur l'interface graphique de l'application.

Pour les mêmes raisons qu'avec Nature Life's, il est assez aisé de comprendre pourquoi il n'y a pas eu lieu d'implémenter un mécanisme de déchargement des plugins. Concrètement, il suffit de s'imaginer qu'une instance d'un plugin est un objet qui exécute sa méthode principale (la méthode `run` dans ce cas). Quand cet objet n'a plus lieu d'exister, la libération des ressources qu'il utilisait se fait automatiquement par le *Garbage Collector*. Ceci ne signifie pas forcément que le plugin n'est plus connu de l'application. Bien au contraire, la classe qui le définit est toujours présente dans un package particulier. Toutefois, lors d'une utilisation ultérieure du même plugin, le chargeur créera à partir de cette classe un nouvel objet qui pourra s'exécuter.

2.6.3 Router Plugins

Le lien entre la structure de Router Plugins et l'architecture générique risque d'être moins trivial que dans les deux cas précédents. Nous ne pourrons d'ailleurs nous contenter de nous baser uniquement sur les schémas de l'architecture (voir pages 25 et 28) de l'application. Nous devrons donc également interpréter, avec un degré d'abstraction assez élevé, les informations qui complètent ces deux schémas. La correspondance est illustrée par la figure 2.4.

En ce qui concerne les composants tiers, nous pouvons les assimiler au noyau IP (qui implémente les mécanismes IP) et aux librairies de fonctions. Rappelons que ces librairies peuvent être utilisées par les différents composants du système et indirectement par des programmes de l'espace de l'utilisateur. Il n'y a donc pas d'objection à ce que nous les considérions comme un composant de notre architecture. Pour accéder à ces librairies de l'extérieur, et notamment configurer différentes instances de plugins, il faut faire usage du *Plugin Manager*, qui offre une interface à l'utilisateur. C'est donc le correspondant au composant GUI de notre modèle.

Comme nous l'avons vu auparavant, un plugin doit avoir été chargé pour être utilisé par le système. Cette opération de chargement est effectuée par le *PCU*, via la commande *modload*. Nous avons également laissé supposer jusqu'ici que le chargement incluait la création d'instances de plugins. Celle-ci se fait également par l'entremise du *PCU*, qui propage les messages de création d'instances en provenance d'autres composants en direction des plugins (voir page 30). Rappelons que les instances sont ici de petites structures de données contenant des paramètres d'application du plugin. Il y a un certain nombre d'autres messages qui peuvent être envoyés aux plugins, tels ceux permettant d'enregistrer une instance auprès de l'*AIU*. Sur le schéma nous n'avons cependant repris que les deux premiers explicités ci-dessus, qui semblent être les plus importants en terme de gestion des plugins. Au vu de ce qui vient d'être dit, il apparaît clairement que les services rendus par le *PCU* sont tout à fait comparables à la plupart des fonctions du chargeur, dans notre modèle générique.

Néanmoins, il a été vu précédemment qu'une autre fonction du chargeur était de pouvoir fournir aux composants qui en ont besoin un pointeur vers les plugins à utiliser. Cela fait

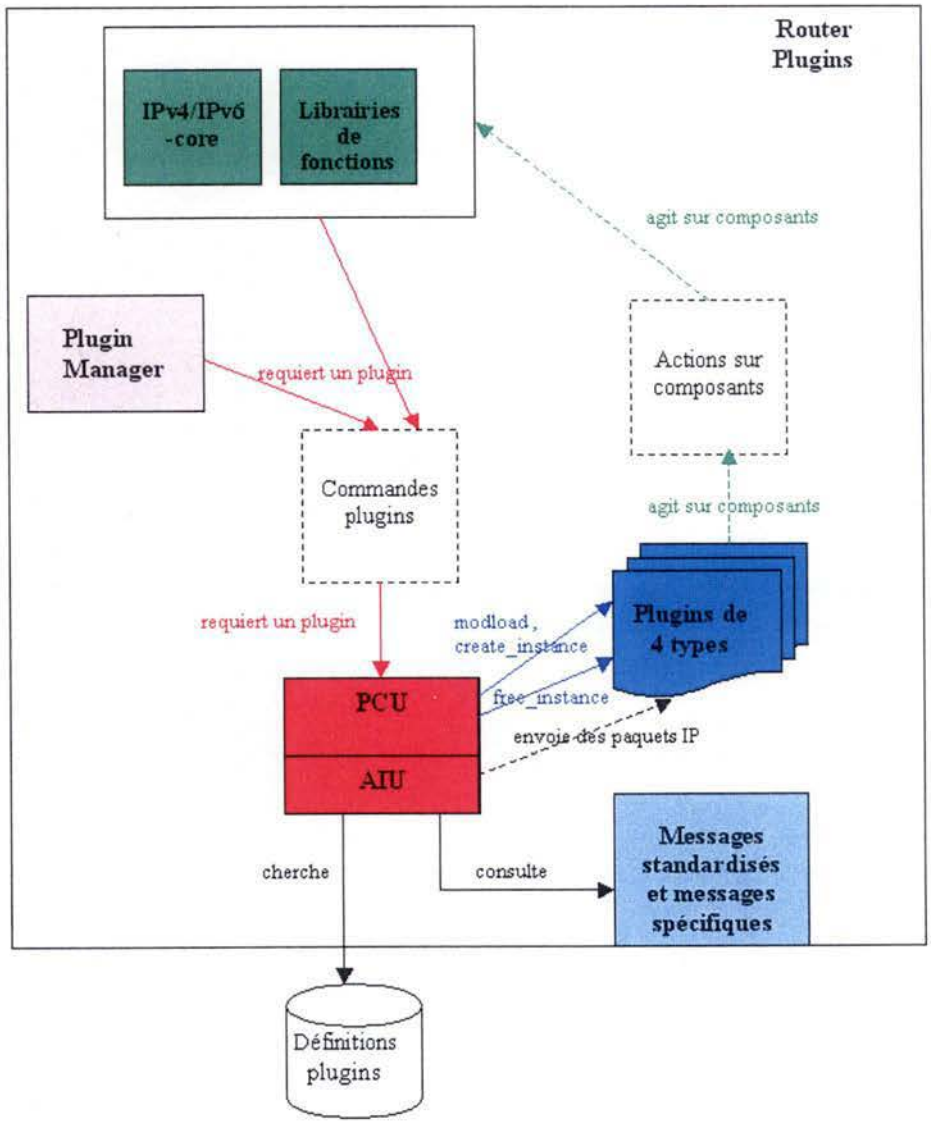


FIG. 2.4 – Correspondance entre l'architecture générique et la structure d'ImageJ

intégralement partie de la tâche de gestion des plugins. Or, il se fait qu'il s'agit précisément du service rendu au noyau IP par le composant *AIU* de Router Plugins. Il ne peut donc, lui aussi, que faire partie de ce que nous avons appelé le chargeur, dans notre architecture générique.

Par conséquent, pour les raisons qui viennent d'être évoquées dans les deux paragraphes précédents, nous considérerons que le chargeur correspond, dans Router Plugins, à l'association des composants *PCU* et *AIU*.

Revenons à présent à l'opération de chargement en elle-même, qui nécessite de pouvoir trouver quelque part les définitions des plugins à charger. Dans les informations dont nous disposons, il n'est nullement fait allusion à un espace particulier contenant ces renseignements. Cependant, comme il est impensable de pouvoir charger des plugins à partir de rien, nous conserverons le composant correspondant tel que nous l'avons présenté dans le modèle.

En ce qui concerne le composant "Interface plugins" de l'architecture générique, il semble être composé de l'ensemble des messages standardisés et des messages spécifiques à certains plugins évoqués à la page 34. En effet, ceux-ci correspondent à l'ensemble des messages auxquels les plugins doivent pouvoir répondre pour fonctionner correctement. C'est précisément la définition abstraite que nous avons donnée du composant "Interface plugins". De manière plus concrète, on peut dire que le *PCU* transforme les demandes des autres composants en direction des plugins en messages présents dans l'ensemble des messages valides. En considérant qu'il consulte cet ensemble de messages, nous justifions le maintien de la flèche intitulée "consulte" dans l'architecture standard.

Nous allons devoir à présent faire preuve d'un peu plus d'abstraction pour établir le lien entre l'exécution proprement dite des plugins du système étudié et ceux du modèle générique. Nous avons pris l'habitude avec les deux cas précédents de considérer que le chargeur lance, quand il le désire, l'exécution de tout ou d'un partie d'un plugin. C'est encore le cas ici, à la différence prêt que les plugins de Router Plugins s'exécutent continuellement à partir du moment où ils sont instanciés. Lorsqu'un plugin doit s'exécuter sur un paquet particulier, ce paquet est en réalité envoyé par le noyau IP. C'est pourquoi nous avons libellé la flèche représentant l'exécution d'un plugin "envoie des paquets IP". Ajoutons encore que c'est le composant *AIU* qui détermine l'instance à utiliser. En considérant la remarque faite auparavant[2], il n'est donc pas faux de faire partir la flèche précitée de ce composant.

Dans le système qui nous préoccupe, l'action des plugins n'a pas directement d'effet sur les composants, puisque ceux-ci opèrent sur des paquets. Cependant, une fois traités, les paquets sont renvoyés au noyau IP qui les acheminera vers les dispositifs du réseau concernés. Avec un peu de flexibilité, nous admettrons que le résultat de l'action des plugins est renvoyé au noyau IP, ce qui justifie le maintien d'une des flèches vertes.

Terminons cette section avec quelques explications sur le déchargement des plugins. Comme nous l'avons vu au premier chapitre (voir page 34), cette action est initiée par le *PCU*, suite à l'envoi au plugin concerné du message "free_instance". C'est pourquoi la flèche correspondante sur la figure 2.4 part du *PCU*. Le plugin gère alors lui-même son déchargement, notamment via l'envoi de messages particuliers au *AIU*.

[2]Il s'agit de la remarque de la fin du quatrième paragraphe de la section "Le chargeur (de plugins)", page 46

2.6.4 Design Patterns

L'analogie entre le *design pattern* exposé dans l'état de l'art et l'architecture proposée dans ce chapitre est assez facile à établir. Certes, le *design pattern* ne reprend pas autant de composants que nous en avons décelés. Rappelons cependant que le but de ce modèle de conception est de mettre l'accent sur les éléments incontournables lors de l'implémentation en orienté-objet d'une application utilisant le principe du plugin. Or, les trois éléments du design pattern correspondent parfaitement, à des degrés d'abstraction différents, aux trois composants-clés de notre architecture générique. Nous admettrons sans discuter que le `PlugInLoader` du *pattern* joue le rôle du chargeur, alors que l'interface `PlugIn` est tout à fait comparable au composant "Interfaces plugins" de l'architecture. Cette interface reprend bien les méthodes que doit définir un plugin concret pour être valide. Enfin, ces mêmes plugins concrets (`ConcretePlugin` dans le pattern), une fois validés par le `PlugInLoader`, correspondent à la description que nous avons faite des "Plugins chargés" du modèle générique. Ces correspondances sont illustrées par la figure 2.5.

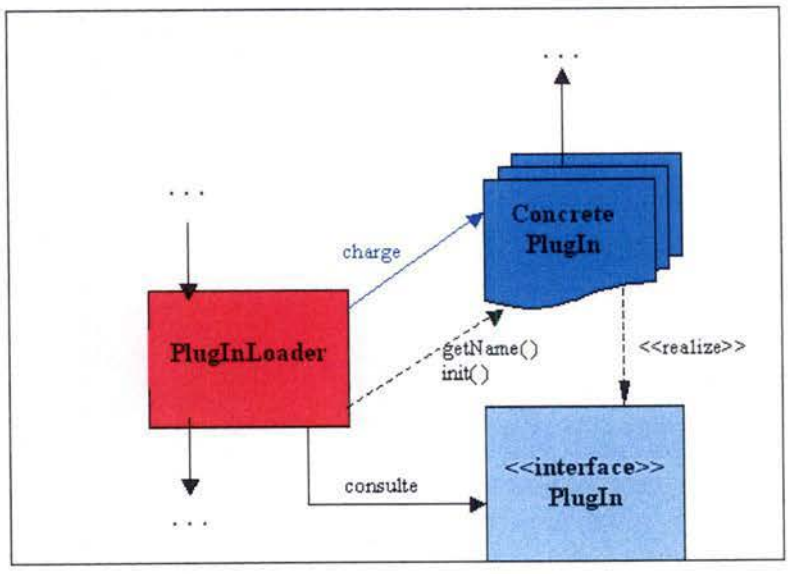


FIG. 2.5 – Correspondance entre l'architecture générique et le *pattern* Plugin

2.7 Les instructions-clés en Java

L'architecture proposée en début de chapitre nous a fait partir d'un niveau d'abstraction très élevé dans notre étude du concept de plugin. Avec le pattern-objet qui vient d'être évoqué à

nouveau, nous nous sommes rapprochés du niveau le plus avancé de la conception d'un logiciel, que constitue le code. Pour terminer notre tour d'horizon, il suffirait donc de donner des indices d'implémentation des concepts évoqués précédemment. C'est ce que nous allons faire à présent en rassemblant les instructions Java permettant de mettre en oeuvre les fonctions des composants principaux de l'architecture. Nous supposerons ici que le lecteur dispose d'une certaine connaissance de la programmation en Java. Le choix de ce langage peut se justifier par deux arguments déjà cités antérieurement. D'une part, nous sommes arrivés à la conclusion que la programmation orientée-objet semblait être un paradigme approprié à la conception d'un système de plugins. D'autre part, deux des trois cas étudiés, ainsi que le système développé pendant le stage et expliqué par après sont conçus en Java. Pour compléter les informations données ci-dessous, il est fortement conseillé de se référer aux spécifications des librairies Java de base [3].

Commençons par rappeler une façon assez simple d'implémenter un chargeur. La première fonction qui doit être remplie est de pouvoir charger un plugin à partir d'une définition, de façon à ce qu'il puisse être utilisé par l'application principale. En Java, ces définitions prennent la forme de fichiers `".class"` qui sont le résultat de la compilation de fichiers `".java"` écrits par les programmeurs tiers. Au début de l'exécution du programme principal, ces définitions ne sont pas connues et les classes correspondantes ne sont donc pas chargées. Pour les charger, il suffit de définir une classe qui étend la classe `java.lang.ClassLoader`. On peut alors appliquer à une instance de ce chargeur la méthode `loadClass(String name)` qui prend en argument le nom de la classe à charger. Pour qu'il trouve cette classe sans problème, il aura suffi au préalable de choisir un répertoire où placer les définitions de plugins, et de l'ajouter au *classpath* du programme. Notons qu'il n'est pas indispensable de ré-implémenter la méthode `loadClass`. Cependant, certains programmeurs expérimentés préféreront en donner une meilleure définition pour la rendre plus efficace ou plus adaptée aux besoins. C'est d'ailleurs le cas dans ImageJ (voir Annexe 7, page 104).

La méthode `loadClass` précitée renvoie un objet de type `Class` que ne constitue en rien une instance de plugin utilisable. Il faut donc créer un objet correspondant à une instance. Ceci se fait très facilement en appliquant la méthode `newInstance()` à l'objet de type `Class`.

Nous avons vu qu'il était indispensable de pouvoir vérifier la validité d'un plugin. Cela signifie que celui-ci doit pouvoir répondre à tous les messages définis dans l'interface à laquelle il se réfère. En Java, et en programmation objet en général, il existe une structure également appelée *interface* qui permet de mettre en oeuvre ce concept. Une interface Java fournit simplement un ensemble de méthodes que toute classe qui l'implémente doit avoir défini. Pour préciser qu'elle se réfère à une certaine interface, une définition de plugin doit, dans son entête, comporter une formule du type `"class MyPlugin implements Plugin"`, où `MyPlugin` est le nom du plugin et `Plugin` le nom de l'interface. Le mot réservé `implements` est l'équivalent du libellé "*realize*" sur le schéma du *design pattern* présenté plus tôt. Pour vérifier la validité d'un plugin, il suffit alors d'utiliser l'instruction `instanceof`. Supposons que l'objet correspondant

[3] http ://java.sun.com/j2se/1.4.2/docs/api/

à l'instance du plugin s'appelle `plug`, l'instruction "`plug instanceof Plugin`" renvoie `true` si le plugin est valide et `false` si non.

En ce qui concerne la gestion des instances de plugins, celle-ci peut se faire à l'aide de structures telles que des tableaux ou des listes. Nous supposons le programmeur assez imaginatif que pour utiliser la structure de données convenant le mieux à son cas.

L'exécution des plugins devient alors très simple puisqu'il suffit d'appeler sur les objets correspondants les méthodes adéquates. Dans certains cas il s'agira d'un appel de méthode lançant l'exécution complète du plugin. Dans d'autres, cela pourra donner lieu à de multiples appels en fonction de la communication prévue avec le plugin.

Rappelons encore qu'il n'est pas nécessaire dans ce cas de définir une opération de déchargement des plugins. En effet, nous venons de voir que toutes les structures utilisées prennent la forme d'objets. Or, dès qu'un objet n'est plus référencé, les ressources utilisées sont libérées par le *Garbage Collector*. C'est donc au programmeur de la structure de données gérant les plugins qu'il revient de bien libérer les instances devenues obsolètes.

Terminons en remarquant qu'il n'existe pas une façon unique de programmer un système de plugins. Par exemple, le chargeur utilisé dans Nature Life's (voir Annexe 3, page 93) comporte quelques petites différences d'implémentation par rapport à ce qui vient d'être dit. Néanmoins, nous pouvons considérer que les pistes données ci-dessus sont dans la plupart des cas suffisantes pour implémenter un système simple.

Chapitre 3

Développements pratiques

Le but de ce chapitre est de présenter les travaux réalisés pendant le stage de fin d'études. Celui-ci a été effectué aux Cliniques Universitaires de Mont-Godinne. Avant de présenter le logiciel d'imagerie médicale qui fut l'objectif du travail, nous verrons quelle est l'utilité de pouvoir utiliser un système de plugins sur une telle application. Nous parlerons également du contexte dans lequel ces travaux ont été effectués, ainsi que les contraintes qui devaient être respectées. Nous exposerons ensuite les résultats finaux des travaux. L'analyse théorique du chapitre précédent ayant été réalisée après le stage, il est peut être intéressant de comparer avec celle-ci le système développé. C'est ce qui fera l'objet de la section finale de ce chapitre.

3.1 Utilité d'un système de plugins dans un logiciel d'imagerie médicale

Plusieurs caractéristiques des logiciels d'imagerie médicale, et plus particulièrement de visualisation et de traitement d'images médicales, peuvent nous aider à comprendre l'intérêt d'un système de plugins.

Tout d'abord, il faut savoir que les images médicales peuvent provenir de différents plateaux techniques d'acquisition, appelés modalités. Parmi celles-ci, citons par exemple la "*Positron Emission Tomography*" (PET), la "*Magnetic Resonance*" (MR) ou encore la "*Computed Tomography*" (CT). Cela implique une certaine diversité du type et du format des images que l'on peut trouver et introduit déjà la nécessité de disposer de plusieurs algorithmes de traitement de ces images.

Ensuite, remarquons qu'un document médical est rarement examiné par un seul médecin. Par exemple, un dossier contenant un certain nombre d'images provenant d'examens d'un patient peut très bien être analysé par un médecin généraliste, un chirurgien et un cardiologue. Or, ces derniers n'utilisent pas forcément les mêmes moyens de traitement des images pour établir leur diagnostic. Cela accroît encore le nombre d'outils dont on doit pouvoir disposer.

Ce qu'il est important de comprendre c'est que tous ces outils nécessaires ne doivent pas se trouver simultanément sur la même machine. En effet, il est par exemple inutile au cardiologue de disposer d'instruments de traitement spécifiques à l'orthopédiste. De plus, inclure tous ces outils dans une même version du logiciel alourdirait fortement le système et en diminuerait les performances.

En admettant que le traitement d'images est gourmand en ressources et que les ordinateurs ne sont pas forcément dédiés à l'exécution d'une seule tâche, l'utilité d'un système de plugins devient évidente. Les outils dont il est question prennent alors la forme de plugins que les médecins peuvent charger à leur guise, en fonction de leurs besoins. Le système est alors utilisé d'une façon plus optimale et on peut même se permettre de personnaliser certains outils en fonction des préférences du médecin. C'est précisément pour ces raisons que la mise en place d'un tel système s'est avérée intéressante pour les responsables du département d'imagerie

médicale de la Clinique de Mont-Godinne.

3.2 Contexte de travail et contraintes à respecter

Le but du stage de fin d'études était d'intégrer au niveau du viewer du logiciel Telemis un système permettant de compiler et d'héberger des plugins traitant des images affichées. Cette tâche est fortement différente de la conception complète d'un logiciel puisque l'existant est déjà bien en place et ne peut être modifié facilement.

Une première remarque à faire à propos de ce travail est qu'aucune étude théorique ni aucun modèle de conception n'avait alors été trouvé. Etant donné que l'objectif à atteindre se rapprochait très fort de ce que propose le logiciel ImageJ, le code de celui-ci est donc devenu la principale source d'informations à analyser. La première tâche à accomplir, qui fut une des plus lentes, fut donc d'éplucher ces sources pour en tirer les parties de code intéressantes en vue d'implémenter un système de gestion de plugins.

Ensuite, remarquons que pour des raisons commerciales de confidentialité évidentes, il était préférable de consulter le moins de sources et le moins de documentation possible sur la conception du logiciel Telemis. Une autre difficulté rencontrée fut donc de cibler le plus possible les questions relatives à l'implémentation de Telemis, afin de respecter au mieux cette confidentialité.

Enfin, la plus grosse contrainte à respecter était le fait qu'il fallait modifier au minimum, voire pas du tout, les sources du programme. Le but véritable devenait donc de greffer à côté de l'application un système de gestion de plugins sans que celui-ci influe sur l'existant logiciel. L'objectif ultime était donc de fournir un petit module qui installe et enlève facilement ce système de plugins sans conséquence sur l'application Telemis. Remarquons également qu'il était préférable que des changements futurs au niveau des sources de Telemis n'influent pas sur notre système de plugins, qui devait continuer à fonctionner malgré ces modifications. Dans cette optique, un autre sous-objectif du travail à accomplir était de fournir aux concepteurs de Telemis une série de recommandations en vue de garantir la longévité du système alors mis en place.

3.3 Le logiciel "Telemis"

3.3.1 Description

Telemis est un système de télé-médecine permettant de consulter et d'échanger des documents médicaux et des images en provenance des modalités. Il a été développé par une

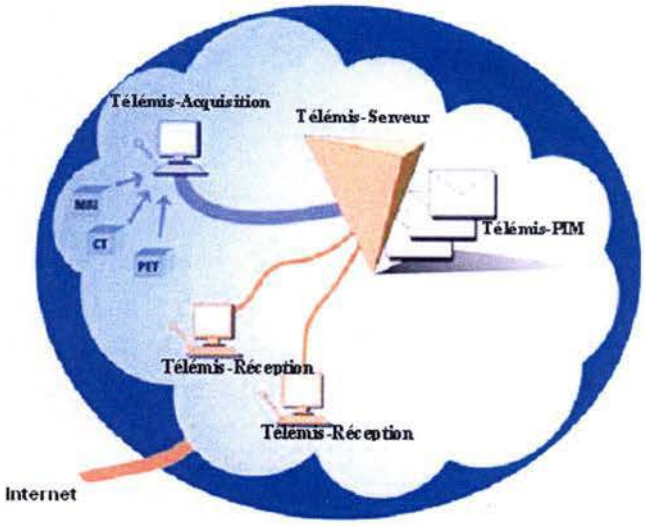


FIG. 3.1 – Architecture à trois composants de Telemis

spin-off de l'Université Catholique de Louvain portant le même nom [1]. L'ensemble logiciel que constitue Telemis est composé de trois modules conçus en Java. Ceux-ci apparaissent sur la figure 3.1.

Le module "Telemis-Acquisition" est comme son nom l'indique responsable de l'acquisition des images en provenance des modalités. Celles-ci sont acheminées à un serveur d'acquisition sous le format DICOM, en utilisant le protocole DICOM [2]. Elles sont alors converties en un format propriétaire correspondant à un document multimédia. L'application d'un système de filtres permet ensuite de déposer ces documents dans des boîtes aux lettres personnalisées ("Telemis-Pim").

Le second module, appelé "Telemis-Serveur" est responsable du stockage des documents Telemis en provenance du module d'acquisition. Chaque destinataire y possède une boîte qui peut être consultée après identification. La consultation correspond en réalité à l'envoi des documents sur le module "Telemis-Réception" implanté sur l'ordinateur du destinataire. Notons que les échanges de documents n'ont pas forcément lieu dans le cadre d'un réseau local. Un médecin identifié peut par exemple relever sa boîte "Telemis-Pim" à partir de l'ordinateur de son domicile pour autant qu'un module de réception y soit installé.

Le dernier module, "Telemis-Réception", nous intéresse plus particulièrement puisque c'est lui qui a fait l'objet de l'implantation du système de plugins. Ce module permet aux utilisateurs authentifiés par une clé d'identification personnelle et un mot de passe d'accéder

[1] http ://www.telemis.be
[2] http ://medical.nema.org/dicom/2003.html

à leur boîte. Ils peuvent alors découvrir la liste des documents sécurisés qui leur sont adressés (voir Annexe 9, page 114). Un simple clic sur le libellé d'un document en permet la consultation. Le module de réception charge alors les images contenues dans ce document, ce qui comprend la décompression, le décryptage et l'affichage des images dans un viewer qui les présente sous forme de galerie. En fonction du type des images (deux ou trois dimensions), celles-ci peuvent apparaître dans des interfaces différentes. La tâche qui nous intéresse a été réalisée au niveau du viewer "2D", illustré à la figure 3.2. De nombreux outils de manipulation et de traitement d'images sont fournis avec le viewer. Parmi eux, citons les possibilités de modifier le contraste ou la luminosité, d'effectuer des zooms, ou encore de sélectionner des régions d'intérêt. Un des objectifs principaux des plugins à implémenter est d'ailleurs d'accroître le nombre de ces outils, en fonction des besoins spécifiques des médecins.

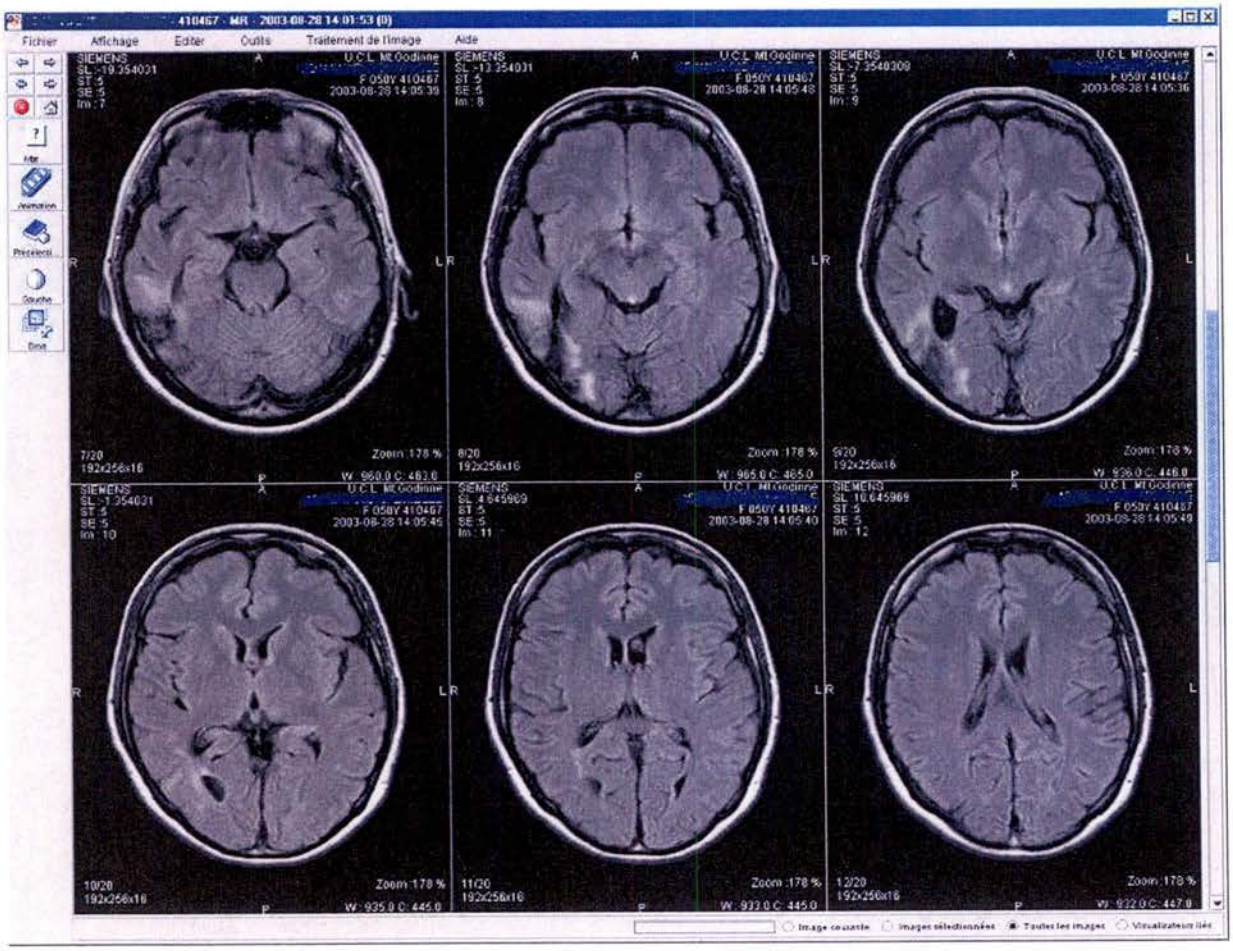


FIG. 3.2 – Viewer 2D de Telemis

3.3.2 Implémentation du module de réception et du viewer

Pour des raisons de confidentialité déjà mentionnées, nous ne présenterons ici que le strict nécessaire pour comprendre la façon dont le système de plugins a été greffé sur Telemis.

Les packages et les classes de l'application sont rassemblés dans le fichier "telemis.jar". Une simple observation du contenu de celui-ci donne déjà une idée suffisante des constituants du module de réception. La découpe en packages est présentée par la figure 3.3.

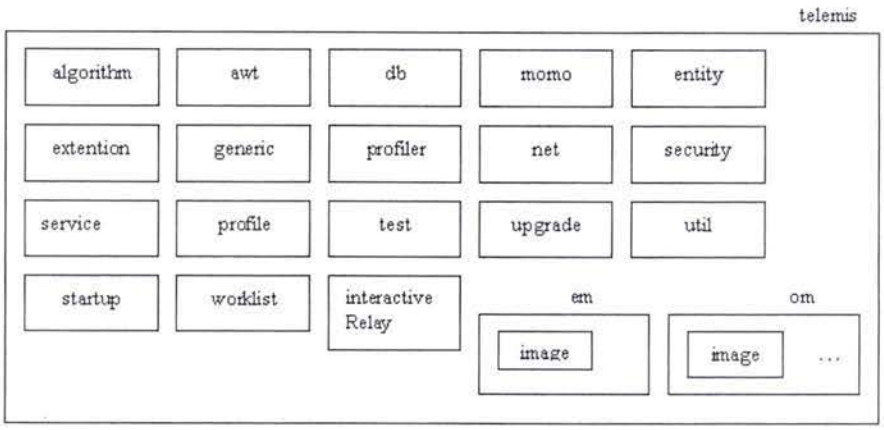


FIG. 3.3 – Les packages du module de réception

Expliquons les rôles de quelques uns d'entre eux :

- **awt** : correspond à une grosse partie de l'interface graphique de l'application.
- **db** : gère la communication avec les bases de données de documents.
- **extention** : permet d'ajouter des fonctionnalités à certains composants du logiciel sans nécessairement en modifier le code.
- **net** : gère les interactions avec le réseau.
- **security** : fournit des mécanismes de sécurité.
- **startup** : gère le démarrage de l'application.
- **om** : ce package gère notamment les différents types d'images affichables par le module de réception. C'est d'ailleurs dans le package `om.image` que sont définis les formats d'images utilisés par Telemis. C'est pourquoi l'on y retrouve également la définition des différents viewers responsables de l'affichage.

Regardons enfin brièvement la structure en terme d'objets du viewer (voir figure 3.4). Ceci nous intéresse dans la mesure où la communication avec les plugins peut se faire uniquement à partir du viewer. En effet, le principal est de savoir comment récupérer des informations dans le viewer et les ré-insérer après traitement par un plugin.

`ImageWindow` est la classe qui correspond au viewer 2D. C'est elle qui manipule le contexte

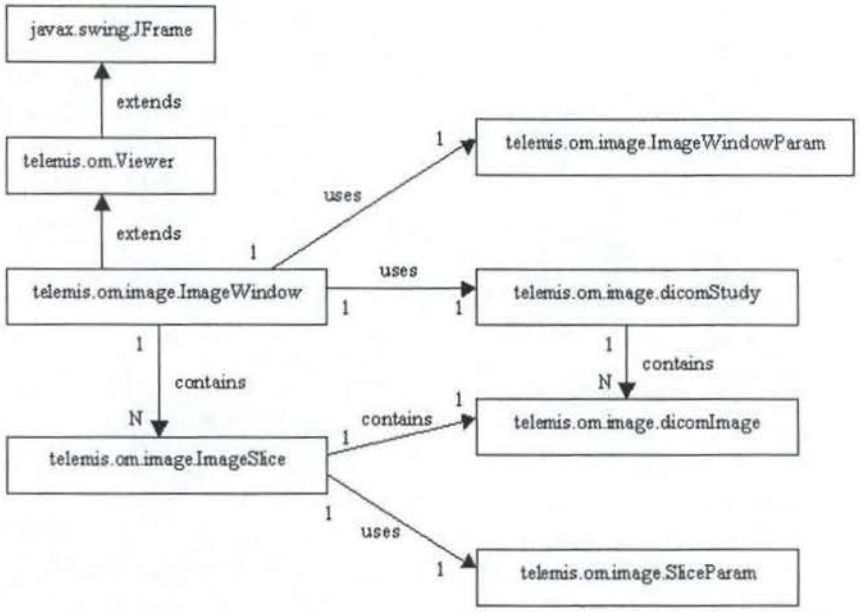


FIG. 3.4 – La structure en objets du viewer 2D

graphique des images et gère les menus et barres d'outils. A chaque `ImageWindow` est associé une instance de `ImageWindowParam` qui contient les caractéristiques du viewer(taille des fenêtres, dimensions des matrices d'affichages, etc ...). Une `ImageWindow` contient un vecteur d'instances d'`ImageSlice`. Chaque `ImageSlice` gère l'affichage d'une `dicomImage`, qui correspond à une tranche dans une image qui en contient plusieurs. A une `ImageSlice` est associée une instance de `SliceParam`, qui en spécifie les caractéristiques (facteurs de zoom, contraste et luminosité, numéro de la tranche, etc ...). Ajoutons encore qu'une `ImageWindow` est liée à une `dicomStudy`. Cette dernière gère les données correspondant aux images contenues dans le document. Elle contient d'ailleurs un certain nombre d'instances de dicomImage, en fonction du nombre de tranches que l'image possède. On a alors fait le tour puisque l'on sait maintenant faire le lien entre le viewer et les données qu'il affiche.

3.4 Résultats

Nous examinerons dans un premier temps l'outil ajouté et ce qu'il permet de faire. Quelques plugins implémentés pendant le stage seront ensuite brièvement présentés. Quelques explications sur l'implémentation du système suivront pour clôturer la section.

3.4.1 L'outil

Avant de présenter l'outil, remarquons qu'un manuel d'instructions sur son fonctionnement et la manière de l'utiliser a été écrit à la fin du stage. Celui-ci se trouve dans sa version intégrale en Annexe 10 (page 115). Les informations livrées dans cette section sont redondantes avec une partie de celles se trouvant dans la manuel. On y trouve notamment tout ce que le programmeur de plugins doit savoir pour mener à bien sa tâche. Les lecteurs désireux de lire ce manuel d'instructions peuvent donc passer cette section qui se veut être une synthèse des informations essentielles sur le fonctionnement du système mis en place.

Rappelons que le but premier du système implémenté est de pouvoir fournir des outils supplémentaires ajoutables à tout moment de l'exécution de Telemis. Ces outils qui prennent la forme de plugins permettent de travailler principalement au niveau de l'image de manière relativement indépendante par rapport à l'application. Cela sous-entend la possibilité de récupérer les données affichées dans le viewer pour que le plugin les traite de manière indépendante. On doit alors pouvoir les ré-introduire dans le viewer pour que les modifications soient éventuellement sauvegardées.

De manière concrète, un plugin de Telemis est en réalité un ensemble de fichiers ".class", résultat de la compilation d'un fichier ".java" contenant le code. A chacun de ces ensembles est également associé un fichier avec une extension ".plugin", indispensable pour la gestion des plugins par le système. Celui-ci contient notamment le nom du plugin à l'intérieur de l'application et le nom des fichiers qu'il utilise. Il est surtout très utile lorsque l'utilisateur désire enlever un plugin de l'application. Son contenu exact est précisé dans le manuel.

Le résultat final du système de gestion de plugins prend la forme d'un kit d'installation dont l'exécution effectue tous les ajouts nécessaires à une utilisation directe du système. Une fois le système installé, un menu "Plugins" s'ajoute automatiquement à la barre de menus principale du viewer. Celui-ci est illustré à la fin du manuel d'instructions. Partons de ce menu pour expliquer les fonctionnalités offertes par le système.

La première fonction du système mis en place (*Load a plugin ...*) permet de charger un plugin pendant l'exécution de l'application, de sorte qu'il soit directement utilisable. Une boite de sélection de fichier permettra de sélectionner sur le disque un fichier ".plugin" à partir duquel le plugin va se charger. Il faudra cependant veiller à ce que tous les fichiers nécessaires à l'exécution du plugin se situe dans le même répertoire. De plus amples informations sur ces contraintes à respecter sont fournies dans le manuel d'instruction. Une fois le plugin chargé, un item de menu libellé avec le nom du plugin s'ajoute au menu des plugins installés, et son exécution peut être lancée par un simple clic.

Le second outil, correspondant à l'item libellé "*Delete a plugin ...*" offre la possibilité de décharger un plugin de sorte qu'il soit rendu inconnu de l'application. Une boite de sélection de fichier permettra de sélectionner le ".plugin" concerné. Une fois le plugin choisi, l'effacement des fichiers inutiles et de l'item correspondant sera automatique.

Remarquons que les deux premières fonctions présentées ci-dessus peuvent très bien être utilisées directement par le médecin. Pour cette raison, elles se distinguent des deux suivantes, plutôt destinées au programmeur de plugins ou à un éventuel gestionnaire de l'application.

Le kit de développement de plugins permet de compiler les plugins à l'intérieur même de l'application. C'est l'objet de la présence de l'item "*Compile and install a plugin ...*". Cette fonction peut s'avérer précieuse pour le programmeur qui teste son plugin, puisqu'il n'est pas obligé de quitter l'application entre deux compilations successives d'un même plugin. Lorsque cet outil est appelé, le programmeur sélectionne le fichier ".java" qui reprend la définition du plugin. Celui-ci est compilé et un éventuel message d'erreur apparaît en cas de problème. Signalons que le fichier ".plugin" ainsi que les autres fichiers utilisés par le plugin doivent se trouver dans le même répertoire que le ".java". Ces contraintes sont détaillées dans le manuel à la section "Compilation des plugins". En cas d'incohérence entre le contenu du fichier de configuration (".plugin") et les fichiers présents dans le répertoire, l'installation du plugin n'est pas achevée avec succès et l'utilisateur en est prévenu. Si tout se passe bien, le plugin est correctement installé et prêt à l'emploi.

La dernière fonction s'adresse particulièrement au programmeur du plugin, pendant la phase de développement de ce dernier. Elle permet de compiler le fichier source dans un environnement autre que celui de Telemis. De cette façon, le programmeur peut résoudre tous les problèmes de compilation sans interférer avec l'application. En cas d'erreurs de compilation, les messages apparaissent dans une fenêtre indépendante. Ajoutons encore que cet outil crée et remplit automatiquement le fichier ".plugin" qui définit en quelque sorte l'environnement du plugin. C'est pour cette raison que l'item de menu correpondant est intitulé "*Build a plugin's environment ...*".

3.4.2 Exemples de plugins

Plusieurs plugins de différents types ont été développés au cours du stage, dans le but de tester l'efficacité du système et de déceler toutes les informations concernant les sources de Telemis utiles au programmeur. Celles-ci sont d'ailleurs regroupées intégralement dans le manuel. Parmi ces plugins figure notamment un inverseur, qui, sans rentrer dans les détails, transforme les pixels les plus foncés (les plus noirs) en pixels clairs (plus blancs), et vice-versa. Un plugin offrant une interface pour envoyer directement par mail l'image sélectionnée a également été créé.

Deux plugins ont également été conçus dans une optique de mise en correspondance d'images. Cette technique, appelée co-registration présente un grand nombre d'applications. Cela est par exemple très utile pour confronter des images provenant de modalités différentes. On peut également s'en servir pour déterminer l'évolution d'une maladie, en comparant des images provenant d'une même modalité mais prises à des moments différents. Les deux plugins dont il est question ici sont à placer dans un cadre de co-registration manuelle, préalable à une éventuelle co-registration automatique.

Le premier, baptisé "*ROI Maker*", permet dans un premier temps de mettre en évidence un ensemble de pixels d'une image se situant en dessous ou au dessus d'une certaine valeur. Concrètement on peut par exemple faire ressortir les pixels les plus clairs d'une image en noircissant tous ceux qui se situent en dessous d'une valeur limite (0 correspondant souvent au noir le plus foncé). Cet ensemble de pixels, que nous appelons alors région d'intérêt (*Region Of Interest - ROI*) peut alors être sauvegardé dans un format propriétaire. On peut par exemple ainsi rassembler une collection de régions d'intérêt de référence.

Le second de ces plugins, appelé "*Import ROI*" offre la possibilité de récupérer une des régions d'intérêt précitées et de la greffer sur une image quelconque. Remarquons que le plugin ajuste lui même la taille de la région d'intérêt pour conserver la cohérence entre les tailles réelles entre les formes représentées par les images.

Signalons pour terminer qu'un exemple illustré d'utilisation du premier de ces plugins est fourni en Annexe 11 (page 129). Le code-source du second figure intégralement en Annexe 12 (page 132).

3.4.3 L'implémentation

Dans cette section, nous regarderons de manière assez générale comment a été implémenté le système de gestion des plugins. Dans un second temps nous expliquerons un peu plus précisément comment a été réglé le problème de dépendances de fichiers, généré par l'effacement des plugins.

Le package `telemis.plugins`

Rappelons pour commencer que le but de ce travail était de greffer un système permettant de gérer des plugins, à côté des sources de Telemis, sans devoir modifier ces dernières. Pour réaliser cela, un fichier nommé "*plugins.jar*" a été ajouté dans le répertoire de Telemis contenant les librairies du programme. Ce fichier d'archives reprend, sous forme d'un ensemble de fichiers "*.class*", la plupart des éléments du système mis en place. L'ensemble de classes dont il est question se situe dans un package `telemis.plugin` qui constitue la greffe à placer au niveau de la découpe en package de l'application (voir figure 3.3, page 66). Regardons quelles sont les classes qu'il contient et leur utilité :

- **PluginClassLoader** : gère le chargement des classes correspondant à des plugins.
- **interface Plugin** : définit la seule et unique méthode que les plugins doivent obligatoirement implémenter. Il s'agit logiquement de la méthode de lancement de l'exécution du plugin (`runit`).
- **Compiler** : gère l'outil de compilation et d'installation d'un plugin. Remarquons que la compilation nécessite l'utilisation d'une librairie java qui se trouve dans un fichier

appelé "*tools.jar*". Tout comme "*plugins.jar*", il doit également être ajouté dans le répertoire des librairies utilisées par Telemis.

- **Debugger** : gère l'outil de déboguage et de création de l'environnement. La même remarque peut être faite que ci-dessus à propos de "*tools.jar*".
- **Extension_Filter** : définit un filtre pour les extensions de fichiers, utilisé lors de la sélection de fichiers correspondant à des plugins.
- **ClassPath_Modifier** : permet de modifier le *classpath* de l'application en cours d'exécution. Cette classe est utilisée lors de l'installation de plugins nécessitant l'utilisation de librairies encore inconnues à ce moment de l'exécution.
- **Error_Editor** : définit la fenêtre utilisée pour l'affichage des messages d'erreur de compilation.
- **MyPluginExtensionBP** : contient notamment la définition du menu des plugins, ainsi que toutes les méthodes qui sont appelées à partir de ceux-ci. Cette classe nécessite quelques explications supplémentaires. Nous avons mentionné auparavant le fait qu'il existait dans Telemis un mécanisme permettant d'étendre certains composants sans en modifier leur source. Pour ce faire il est nécessaire de définir une interface qui doit étendre la classe `telemis.extention.Extention`. Cette interface reprend toutes les méthodes que l'on veut pouvoir appeler à partir de l'extension créée. Dans notre cas cette interface est définie dans le fichier "*MyPluginExtension.class*" que nous avons placé dans le package `telemis.util`. Il s'agit ici en réalité du seul ajout que les gestionnaires du logiciel Telemis doivent apporter aux sources pour que notre système fonctionne. La classe `MyPluginExtensionBP` qui nous concerne ici définit notre extension, qui porte essentiellement sur les ajouts d'un menu et d'items au viewer 2D. Ce dernier, que nous avons déjà évoqué précédemment est en réalité une instance de la classe `telemis.om.image.ImageWindow`, à laquelle on fait référence dans notre extension. Pour que l'extension soit bien prise en compte par l'application, il est également demandé de placer dans un fichier de configuration bien déterminé le nom de l'interface et des classes qui l'implémentent. De cette façon, à chaque démarrage du système, le système de gestion des extensions charge les extensions reconnues.
- **Utils** : reprend un certain nombre de méthodes utilisées par plusieurs des classes précitées.

Nous ne reviendrons pas sur les instructions essentielles à la gestion des plugins puisque nous en avons déjà fait une synthèse au chapitre précédent. Les sources des classes essentielles au mécanisme qui nous intéresse sont cependant fournies en Annexe 13 (page 137).

Remarquons simplement que les librairies *telemis.jar* et *tools.jar* doivent obligatoirement être ajoutées au *classpath* de l'application pour que le système fonctionne correctement. C'est d'ailleurs une des tâches effectuées par le kit d'installation.

La gestion des effacements de plugins

Nous avons vu auparavant qu'il n'est pas forcément nécessaire en Java d'implémenter explicitement un système de déchargement des plugins. C'est une fois de plus le cas ici. En réalité, à chaque fois que l'on presse sur l'item de menu correspondant à un plugin, un nouvel objet est créé afin d'être directement utilisé. Une fois que son exécution s'achève, cet objet qui n'est plus référencé est libéré par le *Garbage Collector*. Cependant, l'item correspondant reste dans le menu des plugins installés et il se peut qu'il n'en soit plus jamais fait usage. De plus, si le nombre de plugins installés s'élève assez rapidement, le médecin peut vite être perdu, ne sachant plus quel outil il doit utiliser. Dans cette optique, il était donc intéressant de créer un mécanisme qui fait disparaître toute trace d'un plugin dans le logiciel. Cela sous-entend non seulement la suppression de l'item de menu correspondant mais également l'effacement des fichiers de librairies et des sources du plugin qui ne sont plus utilisées.

Cet effacement a fait naître une petite difficulté qu'il fallait gérer et qu'il peut être bon d'expliquer, en cas d'utilisation future d'un système similaire. Ce problème provient de deux phénomènes différents. D'une part, la compilation d'un fichier "*.java*" ne donne pas forcément lieu à un seul fichier "*.class*". Par conséquent, certains plugins se composent en réalité d'une classe principale et d'autres classes utilisées par cette dernière. Lors de l'effacement il faudra donc s'assurer d'avoir bien supprimé l'ensemble des fichiers correspondant au plugin. D'autre part, certains fichiers de librairies peuvent être utilisés par plus d'un plugin. Il faut donc s'assurer, lorsqu'on efface un de ces fichiers, qu'aucun autre plugin n'a recours à son utilisation.

Pour parer à cet obstacle, nous avons utilisé un fichier regroupant les dépendances entre les fichiers utilisés par les plugins. Comme nous l'avons vu plus tôt, le fichier "*.plugin*" qui accompagne un plugin stocke l'ensemble des fichiers utilisés par celui-ci. Le but est d'utiliser en parallèle le contenu de ces fichiers "*.plugin*" et du fichier de dépendance, nommé "*dependencies.cfg*". Ce fichier de dépendance compte simplement le nombre de fois qu'un fichier est référencé, c'est-à-dire qu'un plugin en fait usage. Au chargement d'un plugin, chacun des fichiers utilisés par ce dernier (cités dans le fichier "*.plugin*") doit être référencé une fois dans le fichier de dépendance. Cela signifie qu'il doit être ajouté accompagné du chiffre 1 s'il ne s'y trouvait pas encore. S'il y figurait déjà, il suffit alors d'incrémenter d'une unité ce nombre comptabilisant les utilisations. Supposons que nous ayons installé deux plugins. L'un est définit par le fichier "*coucou.class*" et utilise le fichier "*utils.jar*". L'autre prend sa définition dans le fichier "*hello.class*" et fait également usage de "*utils.jar*". A ce moment, le contenu du fichier "*dependencies.cfg*" sera donc le suivant :

```
coucou.class
1
utils.jar
2
hello.class
1
```

Au moment de l'effacement d'un des plugins, supposons le plugin "coucou", la consultation du fichier "*.plugin*" montrera que les fichiers à supprimer sont "*coucou.class*" et "*utils.jar*". Cependant, l'examen du fichier de dépendance révélera la double utilisation de "*utils.jar*". Ce dernier ne devra donc pas être effacé, mais le nombre d'utilisations de celui-ci devra simplement être réduit à 1.

3.5 Comparaison avec les résultats théoriques

La correspondance entre l'architecture générique proposée au chapitre précédent et la structure du système mis en place dans Telemis est assez facile à établir. Elle est illustrée par la figure 3.5.

En ce qui concerne les composants tiers, on peut facilement les associer à l'ensemble des packages présenté à la figure 3.3. On en extraira cependant quelques classes qui correspondent à des composants particuliers de notre modèle. Il est logique qu'aucune flèche ne parte de ce composant, tout comme il est logique qu'aucune n'y arrive. En effet, rappelons que le système de plugin a été greffé au niveau du viewer que nous avons considéré comme une partie de l'interface graphique. Le composant GUI comprend donc la classe `telemis.om.image.ImageWindow` qui définit le viewer 2D et la classe `telemis.plugins.MyPluginExtensionBP` qui apporte à ce viewer les modifications dues au système de gestion des plugins.

La seule façon de recourir à l'utilisation d'un plugin de Telemis est de l'appeler à partir de l'item correspondant dans le menu des plugins installés. C'est pourquoi la flèche rouge part forcément de l'interface graphique. Au préalable il faut évidemment que le plugin ait été chargé, ce qui est la responsabilité de la classe `telemis.plugins.PluginClassLoader`. C'est donc cette classe qui correspond au chargeur dans l'architecture générique.

Nous avons vu que le chargement des plugins se faisait à partir de fichiers "*.class*" situés dans un répertoire bien déterminé. Toutefois l'outil de compilation permet de tout faire d'une fois à partir de fichiers "*.java*". Nous pouvons donc considérer que ces deux types de sources constituent les définitions de plugins. Pour effectuer la première étape du chargement, le chargeur utilise sa méthode `loadClass`, avec pour argument le fichier "*.class*" correspondant au plugin à charger. Il reste alors à créer une instance du plugin, ce qui est fait grâce à la méthode `newInstance`, déjà mentionnée plusieurs fois plus tôt. Une fois ceci fait, il faut encore vérifier la validité de l'instance créée, à partir de l'interface `telemis.plugin.Plugin`. Cette opération est effectuée via l'instruction booléenne `instanceof` dont nous avons également déjà parlé.

La communication avec les plugins se résume à l'appel de la méthode qui en lance l'exécution. Il s'agit de la méthode `runit`. Nous pouvons alors faire la même remarque que pour l'architecture générique. Même si la méthode `runit` n'est pas directement appelée par le `PluginClassLoader`, cette commande ne peut être exécutée que sur l'instance de plugin que

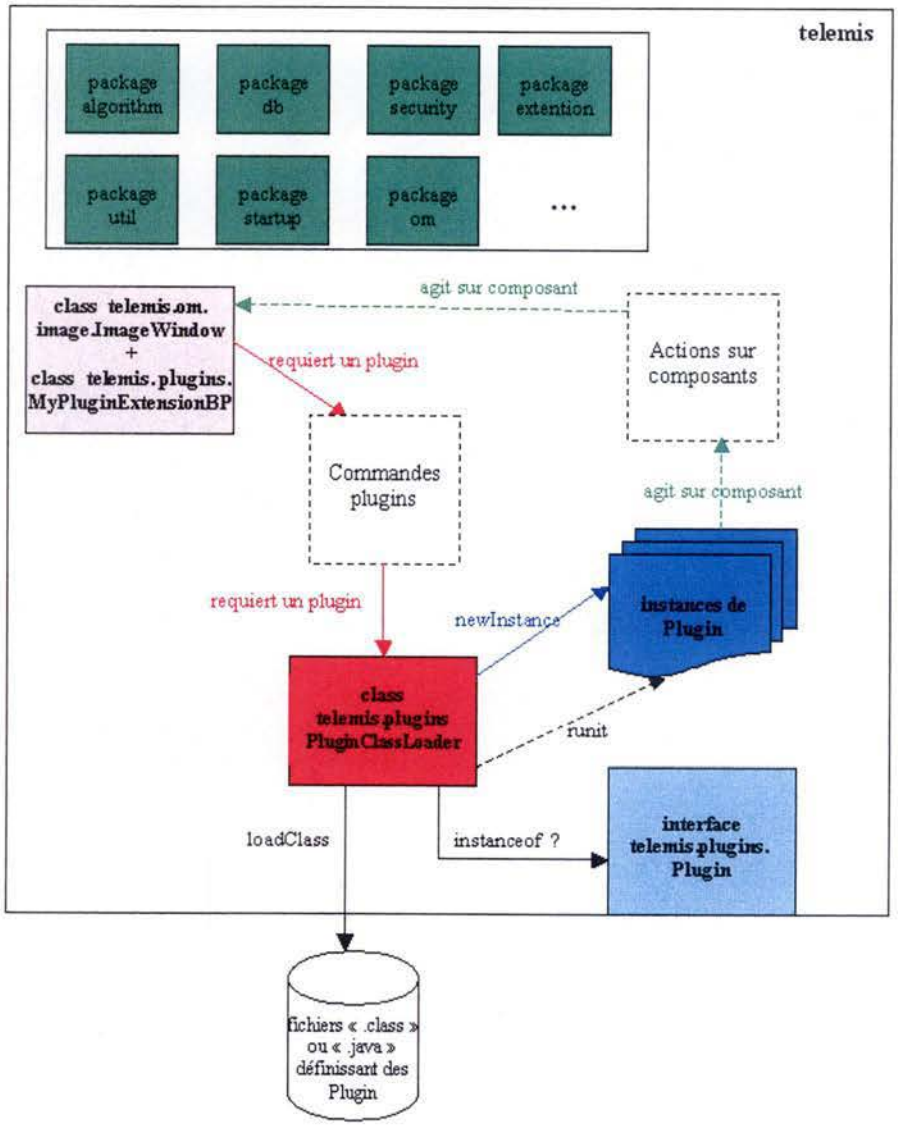


FIG. 3.5 – Correspondance entre l'architecture générique et le système de plugins de Telemis

ce dernier fournit au chargement. C'est pourquoi sur le schéma la flèche en pointillés prend son origine à la boîte représentant le chargeur.

Il est assez aisé de comprendre le maintien de la flèche verte représentant l'action des plugins sur l'interface graphique. En effet, les plugins traitent les images qui se trouvent dans le viewer pour y replacer le résultat de leurs modifications. Leur action sur l'interface graphique est donc indiscutable.

Terminons en remarquant que nous n'avons pas placé de flèche représentant le déchar-

gement des plugins, puisque celui-ci est géré automatiquement par la machine virtuelle Java. Certes, nous avons fait remarquer la présence d'un mécanisme explicite de déchargement. Mais celui-ci agit simplement sur les menus de l'interface et sur les fichiers source. Il ne gère en rien les instances de plugins créées. C'est pourquoi nous ne l'avons pas repris sur le schéma.

Chapitre 4

Discussion

Le but des quelques commentaires qui vont suivre est double. D'une part nous allons essayer d'apporter une vision plus critique du travail qui a été réalisé. D'autre part, nous tenterons de donner des pistes pour un éventuel travail de continuité par rapport à ce qui a été dit dans les chapitres précédents.

Une première remarque qui peut être faite concerne le nombre de cas à partir desquels l'analyse a été réalisée. Il peut en effet sembler peu prudent de tirer des conclusions à partir de l'étude de seulement trois exemples logiciels. Cependant, la contrainte spatiale et le nombre important de points communs faciles à établir entre les cas observés ont poussé à se limiter à cette quantité de données.

Ceci nous amène au second commentaire, concernant la facilité à tirer des conclusions communes aux différents cas. En effet, il fut assez simple de cerner l'ensemble des fonctions que doit remplir un système de gestion de plugins, et de définir des composants en charge de ces fonctions. Nous avons d'ailleurs chercher à transmettre cette simplicité du mécanisme dans l'exposé des résultats. Toutefois, ne perdons pas de vue que nous avons volontairement travaillé à un niveau d'abstraction assez élevé que pour être suffisamment générique. Ceci a d'ailleurs quelque peu simplifié la tâche. En effet, la même réflexion mais à un niveau plus bas, c'est-à-dire plus proche de l'implémentation, aurait vraisemblablement conduit à différents types de résultats. Par exemple, il faut se rendre compte que les concepts simples que nous avons dégagés au niveau des composants présentent de grosses différences d'implémentation, en fonction du langage utilisé pour les mettre en oeuvre. Dans certains cas, cette mise en oeuvre peut d'ailleurs s'avérer assez complexe. Dans cette optique, une tâche qui pourrait encore être réalisée serait l'élaboration de modèles de conception d'un système de plugins, pour la plupart des langages de programmation modernes. Rappelons que nous ne l'avons fait que pour le langage Java.

A propos de l'architecture générique proposée au second chapitre, il faut également remarquer que celle-ci ne se veut surtout pas être un modèle à appliquer dans tous les cas de conception d'un système de gestion des plugins. Le but de celle-ci n'est pas d'offrir un patron que le programmeur doit suivre, mais bien de proposer un panel de composants qui couvre l'ensemble des tâches que doit accomplir un tel système. Des analyses du même genre mais réalisées par d'autres personnes pourraient d'ailleurs proposer des découpes différentes. On pourrait par exemple imaginer une architecture où les fonctions de chargement et d'exécution des plugins sont réalisées par des composants distincts. C'est un choix qui a amené à décider de rassembler ces deux tâches sous la responsabilité d'un composant unique. Et un fois de plus, c'est le degré d'abstraction élevé auquel nous nous sommes placés qui a permis d'effectuer de tels choix.

En ce qui concerne les résultats pratiques exposés dans le troisième chapitre, deux remarques peuvent être faites.

Premièrement, soyons conscients du fait que, même si le système fonctionne correctement et semble répondre aux objectifs, on ne peut malheureusement être sûr qu'il en sera toujours

de même. En effet, Telemis est une application qui ne cesse d'évoluer, et l'on ne peut garantir qu'aucune modification future des sources n'aura d'effets néfastes sur le système de plugins. Toutefois, le mécanisme d'extensions mis en place par les concepteurs de Telemis, et sur lequel notre système se base, semble offrir une flexibilité qui laisse espérer une certaine longévité pour le travail accompli pendant le stage.

Enfin, il faut également remarquer que le nombre de plugins créés et testés au cours du stage reste relativement restreint. Il se pourrait donc qu'il manque dans notre manuel d'instructions certaines informations permettant d'implémenter des outils encore plus spécifiques. Il se pourrait également que notre système manque de robustesse vis-à-vis de certains plugins plus complexes. Il serait par exemple bon d'essayer d'implémenter des modules incluant des routines écrites dans d'autres langages que le Java. Cela impliquerait de devoir utiliser dans nos plugins des systèmes de passerelle tels que le JNI (*Java Native Interface*) [HC00, chapitre 11]. Dans cet ordre d'idées, il serait par exemple intéressant d'essayer d'intégrer dans un de nos plugins une plate-forme de co-registration d'images telle que celle développée au cours d'un projet parallèle à celui qui nous concerne [Vae03]. Ceci ouvrirait la voie à une certaine continuité du travail effectué pendant le stage ...

Conclusion

Ce mémoire avait pour but de proposer une vue globale sur une technique de plus en plus utilisée dans les logiciels modernes, tant elle est synonyme d'économie de ressources.

Pour bien poser le contexte d'utilisation de cette technique, nous avons d'abord tenté d'en donner une définition complète, suivie d'exemples concrets de mise en oeuvre de celle-ci, dans diverses applications. La lecture des présentations de cas a vite fait surgir certains concepts qui semblent inévitables lorsqu'on aborde le principe du plugin. Nous avons alors tenté de structurer quelque peu ces concepts pour définir un certain nombre de composants d'un niveau d'abstraction relativement élevé. La mise en relation de ceux-ci nous a alors permis de proposer une architecture révélatrice des services que doit pouvoir fournir un système de gestion de plugins. Nous avons ensuite considéré comme une validation de notre architecture la mise en correspondance entre celle-ci et les structures des logiciels exposés.

Nous avons par ailleurs remarqué à plusieurs reprises que cette architecture pouvait être traduite de façons très différentes en fonction du type de programmation désiré. A ce sujet nous avons également conclu que la programmation orientée-objet semblait être particulièrement appropriée, notamment parcequ'elle propose une solution directe à la problématique de l'interface, soulevée dans les commentaires de l'architecture.

Ensuite, une synthèse du travail réalisé pendant le stage a été rédigée. Elle nous aura permis dans un premier temps de souligner à nouveau l'utilité de la technique du plugin dans un champ d'application bien spécifique. L'essentiel des résultats concrets des travaux a alors été présenté. Une brève analyse a posteriori de ceux-ci nous a enfin conduit à l'établissement de l'analogie entre les mises en oeuvre pratiques et les développements théoriques d'après-stage. Considérons que nous avons de cette façon bouclé notre tour d'horizon.

Je tiens, pour conclure, à souligner l'aspect enrichissant du stage à la clinique de Mont-Godinne. D'une part, il aura permis de donner un premier aperçu du travail en projet tel qu'il l'est souvent dans la vie professionnelle d'un informaticien. D'autre part, il aura suscité chez moi un grand intérêt, de par la nature de son domaine d'application. Comment pourrait-il d'ailleurs en être autrement, lorsque l'informatique se met au service des sciences humaines, et plus particulièrement de la médecine ...

Bibliographie

[AP] H. Adiseshu and G. Parulkar. SSP : A State Setup Protocol. to be published.

[Bai01] Werner Bailer. Writing imagej plugins - a tutorial, 2001.
http ://webster.fhs-hagenberg.ac.at/staff/burger/ImageJ/tutorial/.

[Bon00] Olivier Bonaventure. Cours de Téléinformatique et réseaux, chapitre 'Fonctionnement d'un routeur ip', FUNDP, 2000.
http ://enligne.info.fundp.ac.be/cours/IHDC2109_0101061200_obo_WCO_N.zip.

[DDPP98] Dan Decasper, Zubin Dittia, Guru M. Parulkar, and Bernhard Plattner. Router plugins : A software architecture for next generation routers. In *SIGCOMM*, pages 229–240, 1998.
http ://citeseer.nj.nec.com/decasper98router.html.

[DH95] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6), Specification, RFC 1883*, December 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[HC00] Cay S. Horstmann and Gary Cornell. *Au coeur de Java 2*, volume 2 : Fonctions avancées. Campus Press France, 2000.

[LB01] D. Le Berre. Java Tip 113 : Identify subclasses at runtime. *Java World*, 2001.
http ://www.javaworld.com/javaworld/javatips/jw-javatip113.html.

[May02] J. Mayer. Graphical user interfaces composed of plug-ins, 2002.
http ://citeseer.nj.nec.com/mayer02graphical.html.

[MCFZ99] W. Meira, Jr., M. Cesário, R. Fonseca, and N. Ziviani. Integrating www caches and search engines, 1999.
http ://citeseer.nj.nec.com/meira99integrating.html.

[MMS02] J. Mayer, I. Melzer, and F. Schweiggert. Lightweight plug-in-based application development, 2002.
http ://citeseer.nj.nec.com/mayer02lightweight.html.

[ST02] A. Shalloway and J.R. Trott. *Design Patterns Explained : A New Perspective on Object-Oriented Design.* Addison-Wesley, 2002.

[Vae03] Olivier Vaesen. Elaboration d'une plateforme orientée-objet dédiée à la coregistration d'images médicales. Master's thesis, Facultés Universitaires Notre-Dame de la Paix de Namur, Institut d'Informatique, 2003.

[Völ99] M. Völter. Pluggable Components - A Pattern for Interactive System Configuration. In *Proceedings of the Fourth European Conference on Pattern Languages of Programming and Computing, EuroPLoP '99, Bad Irsee*, Germany, 1999.

[Web] Reference Document Web. The dive plugin interface. http ://citeseer.nj.nec.com/meira99integrating.html.

[Wir95] N. Wirth. A plea for lean software. *IEEE Computer 28*, pages 64–68, 1995.

[Zea93] L. Zhang and et al. RSVP : A New Resource ReSerVation Protocol. *IEEE Network Magazine*, Vol. 7(No. 5), September 1993.

Annexes

Annexe 1

Les cinq types de bêtes de base de Nature Life's

Coccinelle

Je suis à bon-dieu, mais j'aime pas tendre mes joues.

Hobbies : Météorologie divinatoire

Distance de vue : **50** Déplacement : **Air**

Angle de vue : **150°** Vitesse maximale : **5**

Cochon

Chez moi tout est bon.

Hobbies : Thalassothérapie

Distance de vue : **70** Déplacement : **Terre**

Angle de vue : **70°** Vitesse maximale : **5**

Eléphant

Plus c'est gros, meilleur c'est.

Hobbies : Pompier volontaire

Distance de vue : **80** Déplacement : **Terre**

Angle de vue : **100°** Vitesse maximale : **5**

Hamster

Je ne sens peut-être pas toujours très bon, mais j'ai un joli sourire.

Hobbies : Courir sur place dans ma grande roue

Distance de vue : **70**

Angle de vue : **120°**

Déplacement : **Terre**

Vitesse maximale : **5**

Zèbre

C'est pas parce que je suis habillé comme ça, que je suis un tôlard.

Hobbies : Manger un bon steak de cheval

Distance de vue : **60**

Angle de vue : **100°**

Déplacement : **Terre**

Vitesse maximale : **8**

Annexe 2

Exemple d'implémentation d'un plugin Beast de Nature Life's

```
package beasts;

import java.io.*;
import java.awt.*;
import java.awt.geom.*;
import fr.unice.bertuli.util.Calculation;
import fr.unice.natureLife.pluginSDK.*;

  /**
   *Classe représentant un éléphant, c'est un plugin
   *qui doit être placé dans le répertoire
   *"fr/unice/natureLife/plugins/beasts" pour être pris en compte
   *par NatureLife's.
   *
   *@author Bertuli Roland
   *@author Campanella Fabrice
   */

public class BeastElephant extends Beast implements Serializable {

    /**
     * Constructeur créant une instance de BeastElephant.
     */
    public BeastElephant() {
    setMaxSpeed(5);
    setCourseNb(21);
    setViewDistance(80);
    setFrontDistance(9);
    // Animal marchant
    setFly(false);
    }

    /**
     *Renvoie le nom de l'espèce.
     *
     *@return
     *"Eléphant".
     */
    public String getName() {
    return "Eléphant";
    }
```

```
    /**
     *Renvoie des commentaires sur l'espèce.
     *
     *@return
     *les informations sur l'espèce.
     */
    public String getAbout() {
    return "Espèce: ELEPHANT"
        + "\n\nPlus c'est gros, meilleur c'est."
        + "\n  Hobbies: Pompier volontaire"
        + "\n\nBertuli Roland\nCampanella Fabrice";
    }

    /**
     *Renvoie le nom du fichier qui contient la photo
     *de l'éléphant.
     *
     *@return
     *nom du fichier image.
     */
    public String wherePhoto() {
    return "BeastElephant.gif";
    }

    /**
     *Dessine l'éléphant.
     *
     *@param g le contexte graphique
     */
    public void drawYou(Graphics g) {
    // Dessin de la cible (si elle existe)
    getBrain().drawTarget(g);
        Graphics2D g2d = (Graphics2D)g;
        AffineTransform at = new AffineTransform();
        at.setToRotation(Calculation.degToRad(direction), x, y);
        AffineTransform ati = null;
        try {
        ati = at.createInverse();
        }
        catch(NoninvertibleTransformException e) {}
        g2d.transform(at);
    g2d.setColor(Color.gray);
    g2d.fillOval(x - front, y - front*2/3, front*2, front*3/2);
    g2d.fillOval(x + front/2, y - front*2/3, front/2, front/2);
    g2d.fillOval(x + front/2, y + front/2, front/2, front/2);
    g2d.fillRect(x + front/2, y - front/4, front, front/2);
        g2d.transform(ati);
    }
}
```

Exemple d'implémentation d'un plugin Behaviour de Nature Life's

```
package behaviours;

import java.io.*;
import java.util.*;
import fr.unice.bertuli.util.Calculation;
import fr.unice.natureLife.area.*;
import fr.unice.natureLife.pluginSDK.*;

  /**
   *Classe représentant un comportement qui évite les obstacles,
   *c'est un plugin qui doit être placé dans le répertoire
   *"fr/unice/natureLife/plugins/behaviours" pour être pris
   *en compte par NatureLife's.
   *
   *@author Bertuli Roland
   *@author Campanella Fabrice
   */

public class BehaviourAvoid extends Behaviour implements Serializable {

    /**
     *Donne le nom du comportement.
     *
     *@return
     *"Eviter obstacles".
     */
    public String getName() {
        return "Eviter obstacles";
    }

    /**
     *Donne des informations sur le  comportement.
     *
     *@return
     *les informations du comportement.
     */
    public String getAbout() {
    return "Comportement: EVITER OBSTACLES"
        + "\n\n Evite les caps qui vont vers un obstacle"
        + "\ndans le champ de vision de la bête."
        + "\n\nBertuli Roland\nCampanella Fabrice";
    }

     /**
     *Donne un tableau de valeur représentant les valeurs de préférence
     *de direction (en terme de changement de cap).
```

```
 *
 *@return
 *le tableau de recommandation de cap.
 */
public double[] giveCourses() {
// Drapeau repérant si on n'a pas encore rencontré
// un cap repérant un obstacle
boolean noObstacle = true;
int nbCourses = beast.getCourseNb();
ArrayList badCourses = brain.obstacleSeen();
Iterator it = badCourses.iterator();
while(it.hasNext()) {
    // Création et addition d'un tableau
    // pour chaque cap visant un obstacle
    // au tableau de "recommandation" des caps
    // à prendre
    int obstacleCourse = ((Integer)it.next()).intValue();
    double[] tabTemp = new double[nbCourses];
    for (int i=0; i<nbCourses; i++) {
    // On augmente le poids d'acceptation du cap
    // de 5. Cette valeur est arbitraire
    // Nous renivellerons pour s'approcher d'un total
    // de pondération de 100 sur les "acceptations finales"
    tabTemp[i] = Math.abs(i - obstacleCourse) * 5;
    // Si on n'a pas encore rencontré un "cap à obstacle"
    // on prend ces recommandations
    if (noObstacle) {
        coursePossibilities[i] = tabTemp[i];
    }
    // Sinon on met les informations en commun en prenant
    // les recommandations minimales pour chaque conseil
    // et en les affectant aux recommandations finales
    else {
        if (coursePossibilities[i] > tabTemp[i])
        coursePossibilities[i] = tabTemp[i];
    }
    }
    noObstacle = false;
}
// on avance au hasard
if (noObstacle) {
    for (int i=0; i<nbCourses; i++) {
    coursePossibilities[i] = 100 / nbCourses;
    }
}
// Sinon nivellons les poids pour avoir un total de 100
else {
    int totalPoids = 0;
    for (int i=0; i<nbCourses; i++) {
```

```
        totalPoids += coursePossibilities[i];
        }
        double coef = 100.0 / totalPoids;
        for (int i=0; i<nbCourses; i++) {
        coursePossibilities[i] = coursePossibilities[i] *  coef;
        }
    }
    return coursePossibilities;
    }
}
```

Annexe 3

Code source du PluginLoader de Nature Life's

```
package fr.unice.natureLife.plugins;

import java.io.*;
import java.net.*;
import java.util.*;
import fr.unice.natureLife.pluginSDK.*;

  /**
   *La classe PluginLoader s'occupe de charger des classes
   *de plugins situées dans un répertoire donné et correspondant à un
   *certain type.
   *
   *@author Bertuli Roland
   *@author Campanella Fabrice
   */

public class PluginLoader {

    /**
     *Répertoire contenant les plugins
     */
    private String dir;

    /**
     *Classe des plugins
     */
    private Class clazz;

    /**
     *Construit et initialise un chargeur de plugins.
     *
     *@param dir répertoire des plugins
     *@param clazz classe des plugins
     */
    public PluginLoader(String dir, Class clazz) {
    this.dir = dir;
    this.clazz = clazz;
    }

    /**
     *Permet de repérer des plugins de type 'clazz' présents dans un
     *répertoire 'rep'. Renvoie en résultat un ArrayList contenant les
     *instances des plugins chargés et un autre Arraylist contenant les noms
```

```
 *(sous forme de String) des plugins. Il y a une correspondance entre
 *les deux ArrayList instancesPlugins et nomsPlugins.
 *
 *@return
 *un ArrayList qui contient les instances de plugins cherchées.
 */
public ArrayList loadsPlugins() {
// On regarde si le répertoire contenant les plugins est dans le
// classpath et on retourne l'URL. getRessource() utilise le ClassLoader
// de la classe courante (primordial class loader, le class loader par
// defaut). On aurait pu écrire : getClass().getRessource(rep). En
// faisant comme ça pour lire le contenu du répertoire
ClassLoader cl = PluginLoader.class.getClassLoader();
URL url = PluginLoader.class.getResource(dir);
if (url == null) {
    System.err.println("Répertoire de plugins specifié n'est pas correct");
    return new ArrayList();
}
return loadsClasses(url, dir, clazz);
}

/**
 *Charge les classes de plugins d'un répertoire.
 *
 *@param url URL du répertoire de plugins
 *@param dir répertoire de plugins
 *@param clazz classe des plugins
 *@return
 *les plugins chargés sous forme d'ArrayList.
 */
private ArrayList loadsClasses(URL url, String dir, Class clazz) {
// Contiendra les plugins cherchés
ArrayList plugins = new ArrayList();
File directory = new File(url.getFile());
if (directory == null || !directory.isDirectory()) {
    System.err.println("Le répertoire specifié n'est pas correct : " +
            url);
    return plugins;
}
String[] list = directory.list();
// On trie pour que les plugins apparaissent dans le menu
// par ordre alphabétique
Arrays.sort(list);
for (int i = 0; i < list.length; i++) {
    // Un plugin ne peut être une classe interne
    if (list[i].indexOf('$') == -1 && list[i].endsWith(".class")) {
    // Le premier membre est le package.
    String className = dir.replace(File.separatorChar , '.')  + "."
        + list[i].substring(0,list[i].indexOf('.'));
```

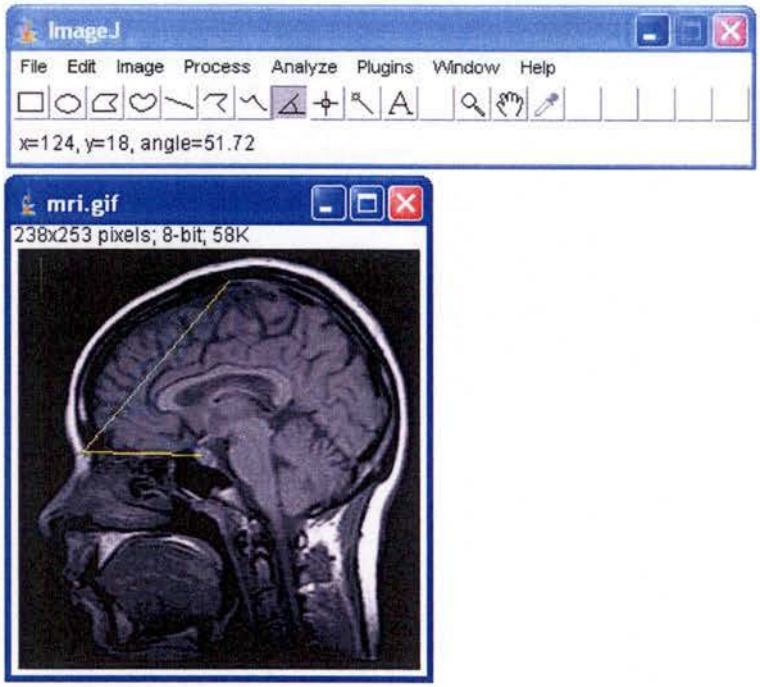
```
    Plugin plugin = createInstance(className);
    if(plugin != null)
        plugins.add(plugin);
    }
}
return plugins;
}

/**
 *Retourne une instance de plugin.
 *
 *@param className nom de la classe à charger
 *@return
 *une instance de la classe ou null si problème.
 */
public Plugin createInstance(String className) {
try {
    // C'est ici que se passe le chargement de la classe!
    Class c = Class.forName(className);
    Plugin plug = null;
    try {
    // On crée une instance de la classe
    plug = (Plugin)c.newInstance();
    }
    catch (ClassCastException e) {
    //Le fichier n'est pas un plugin 'Plugin'
    System.err.println("Le plugin " + className +
              " n'est pas un Plugin");
    return null;
    }
    catch (InstantiationException e ) {
    System.err.println("Le plugin " + className +
              " ne peut pas etre instancié");
    return null;
    }
    catch (IllegalAccessException e ) {
    System.err.println("Le plugin " + className +
              " est interdit d'accés");
    return null;
    }
    catch (NoClassDefFoundError e ) {
    System.err.println("Le plugin " + className +
              " n'est pas un Plugin correct");
    return null;
    }

    // Ruse pour tester si plug est compatible avec la classe clazz
    if (clazz.isInstance(plug))
    return plug;
```

```
        else {
        return null;
        }

    }
    catch (ClassNotFoundException e) {
        System.err.println("Le plugin " + className +
                   " est introuvable");
        return null;
    }
    }
}
```

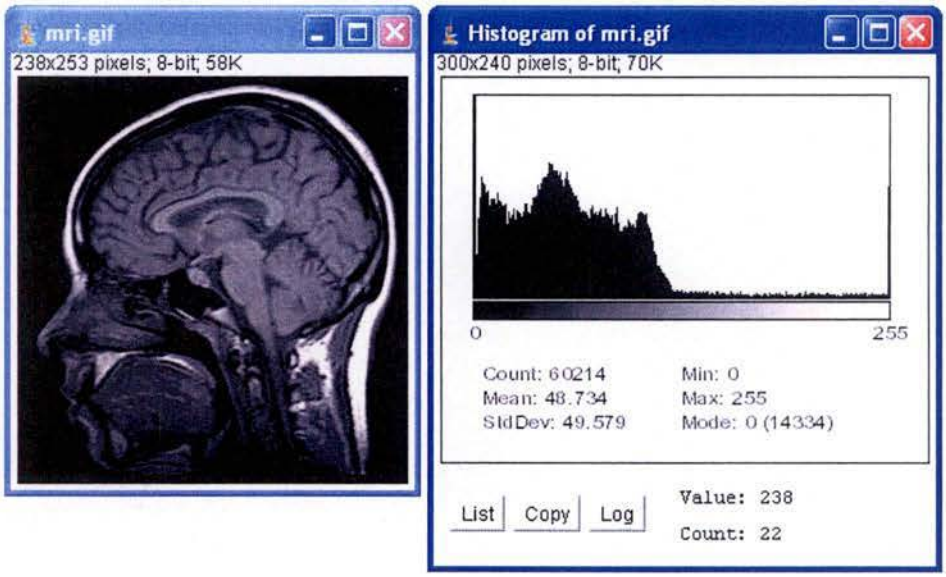
Annexe 4

Quelques fonctionnalités d'ImageJ

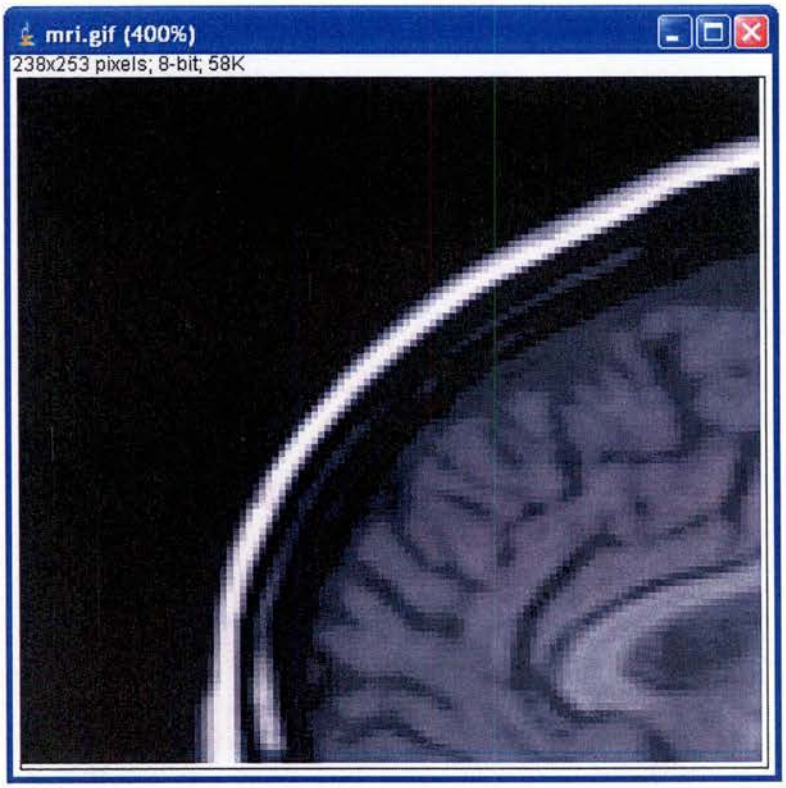
Calcul d'angle



Histogramme de densité



Zoom



Annexe 5

Sources des interfaces `PlugIn` et `PlugInFilter` d'ImageJ

```
package ij.plugin;

/** Plugins that acquire images or display windows should
    implement this interface. Plugins that process images
    should implement the PlugInFilter interface. */
public interface PlugIn {

    /** This method is called when the plugin is loaded.
        'arg', which may be blank, is the argument specified
        for this plugin in IJ_Props.txt. */
    public void run(String arg);

}
```

```
package ij.plugin.filter;
import ij.*;
import ij.process.*;

/** ImageJ plugins that process an image should implement this interface. */
public interface PlugInFilter {

    /** This method is called once when the filter is loaded. 'arg',
        which may be blank, is the argument specified for this plugin
        in IJ_Props.txt. 'imp' is the currently active image.
        This method should return a flag word that specifies the
        filters capabilities. */
    public int setup(String arg, ImagePlus imp);

    /** Filters use this method to process the image. If the
        SUPPORTS_STACKS flag was set, it is called for each slice in
        a stack. ImageJ will lock the image before calling
        this method and unlock it when the filter is finished. */
    public void run(ImageProcessor ip);



    /** Set this flag if the filter handles 8-bit grayscale images. */
    public int DOES_8G = 1;
    /** Set this flag if the filter handles 8-bit indexed color images. */
    public int DOES_8C = 2;
    /** Set this flag if the filter handles short images. */
```

```
    public int DOES_16 = 4;
    /** Set this flag if the filter handles float images. */
    public int DOES_32 = 8;
    /** Set this flag if the filter handles RGB images. */
    public int DOES_RGB = 16;
    /** Set this flag if the filter handles all types of images. */
    public int DOES_ALL = DOES_8G+DOES_8C+DOES_16+DOES_32+DOES_RGB;
    /** Set this flag if the filter wants its run() method to be
        called for all the slices in a stack. */
    public int DOES_STACKS = 32;
    /** Set this flag if the filter wants ImageJ, for non-rectangular
        ROIs, to restore that part of the image that's inside the bounding
        rectangle but outside of the ROI. */
    public int SUPPORTS_MASKING = 64;
    /** Set this flag if the filter makes no changes to the pixel data. */
    public int NO_CHANGES = 128;
    /** Set this flag if the filter does not require undo. */
    public int NO_UNDO = 256;
    /** Set this flag if the filter does not require that an image be open. */
    public int NO_IMAGE_REQUIRED = 512;
    /** Set this flag if the filter requires an ROI. */
    public int ROI_REQUIRED = 1024;
    /** Set this flag if the filter requires a stack. */
    public int STACK_REQUIRED = 2048;
    /** Set this flag if the filter does not want its run method called. */
    public int DONE = 4096;

}
```

Annexe 6

Exemples d'implémentation de plugins d'ImageJ

Un plugin simple

```
import java.awt.*;
import ij.*;
import ij.gui.*;
import ij.process.*;
import ij.plugin.PlugIn;

/** This a prototype ImageJ plugin. */
public class Red_And_Blue implements PlugIn {

    public void run(String arg) {
        int w = 400, h = 400;
        ImageProcessor ip = new ColorProcessor(w, h);
        int[] pixels = (int[])ip.getPixels();
        int i = 0;
        for (int y = 0; y < h; y++) {
            int red = (y * 255) / (h - 1);
            for (int x = 0; x < w; x++) {
                int blue = (x * 255) / (w - 1);
                pixels[i++] = (255 << 24) | (red << 16) | blue;
            }
        }
        new ImagePlus("Red and Blue", ip).show();
    }
}
```

Un plugin filtre

```
import ij.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;
import java.awt.*;

/** This sample ImageJ plug-in filter inverts 8-bit images.
*/

public class Inverter_ implements PlugInFilter {

    public int setup(String arg, ImagePlus imp) {
        if (arg.equals("about"))
            {showAbout(); return DONE;}
        return DOES_8G+DOES_STACKS+SUPPORTS_MASKING;
    }

    public void run(ImageProcessor ip) {
        byte[] pixels = (byte[])ip.getPixels();
        int width = ip.getWidth();
        Rectangle r = ip.getRoi();
        int offset, i;


        for (int y=r.y; y<(r.y+r.height); y++) {
            offset = y*width;
            for (int x=r.x; x<(r.x+r.width); x++) {
                i = offset + x;
                pixels[i] = (byte)(255-pixels[i]);
            }
        }
    }

    void showAbout() {
        IJ.showMessage("About Inverter_...",
            "This sample plug-in filter inverts 8-bit images. Look\n" +
            "at the 'Inverter_.java' source file to see how easy it is\n" +
            "in ImageJ to process non-rectangular ROIs, to process\n" +
            "all the slices in a stack, and to display an About box."
        );
    }
}
```

Annexe 7

Parties de code intéressantes de `ij.IJ` et `ij.io.PluginClassLoader`

ij.IJ

```
package ij;
import ij.gui.*;
import ij.process.*;
import ij.text.*;
import ij.io.*;
import ij.plugin.*;
import ij.plugin.filter.*;

...

    /** Runs the specified plugin and returns a reference to it. */
    public static Object runPlugIn(String className, String arg) {
        return runPlugIn("", className, arg);
    }

    /** Runs the specified plugin and returns a reference to it. */
    static Object runPlugIn(String commandName, String className, String arg) {
        if (IJ.debugMode)
            IJ.log("runPlugin: "+className+" "+arg);
        // Use custom classloader if this is a user plugin
        // and we are not running as an applet
        if (!className.startsWith("ij") && applet==null) {
            boolean createNewClassLoader = altKeyDown();
            return runUserPlugIn(commandName, className, arg, createNewClassLoader);
        }
        Object thePlugIn=null;
        try {
            Class c = Class.forName(className);
            thePlugIn = c.newInstance();
            if (thePlugIn instanceof PlugIn)
                ((PlugIn)thePlugIn).run(arg);
            else
                runFilterPlugIn(thePlugIn, commandName, arg);
        }

        catch (ClassNotFoundException e) {
            if (IJ.getApplet()==null)
                log("Plugin not found: " + className);
        }
        catch (InstantiationException e) {log("Unable to load plugin (ins)");}
        catch (IllegalAccessException e) {log("Unable to load plugin (acc)");}
```

```
    return thePlugIn;
}

static void runFilterPlugIn(Object theFilter, String cmd, String arg) {
    ImagePlus imp = WindowManager.getCurrentImage();
    int capabilities = ((PlugInFilter)theFilter).setup(arg, imp);
    if ((capabilities&PlugInFilter.DONE)!=0)
        return;
    if ((capabilities&PlugInFilter.NO_IMAGE_REQUIRED)!=0)
        {((PlugInFilter)theFilter).run(null); return;}
    if (imp==null)
        {IJ.noImage(); return;}
    if ((capabilities&PlugInFilter.ROI_REQUIRED)!=0 && imp.getRoi()==null)
        {IJ.error("Selection required"); return;}
    if ((capabilities&PlugInFilter.STACK_REQUIRED)!=0 && imp.getStackSize()==1)
        {IJ.error("Stack required"); return;}
    int type = imp.getType();
    switch (type) {
        case ImagePlus.GRAY8:
            if ((capabilities&PlugInFilter.DOES_8G)==0)
                {wrongType(capabilities); return;}
            break;
        case ImagePlus.COLOR_256:
            if ((capabilities&PlugInFilter.DOES_8C)==0)
                {wrongType(capabilities); return;}
            break;
        case ImagePlus.GRAY16:
            if ((capabilities&PlugInFilter.DOES_16)==0)
                {wrongType(capabilities); return;}
            break;
        case ImagePlus.GRAY32:
            if ((capabilities&PlugInFilter.DOES_32)==0)
                {wrongType(capabilities); return;}
            break;
        case ImagePlus.COLOR_RGB:
            if ((capabilities&PlugInFilter.DOES_RGB)==0)
                {wrongType(capabilities); return;}
            break;
    }
    int slices = imp.getStackSize();
    boolean doesStacks = (capabilities&PlugInFilter.DOES_STACKS)!=0;
    if (!imp.lock())
        return; // exit if image is in use
    imp.startTiming();
    IJ.showStatus(cmd + "...");
    ImageProcessor ip;
    ImageStack stack = null;
    if (slices>1)
        stack = imp.getStack();
```

```
int[] mask = null;
float[] cTable = imp.getCalibration().getCTable();
if (slices==1 || !doesStacks) {
    ip = imp.getProcessor();
    mask = imp.getMask();
    if ((capabilities&PlugInFilter.NO_UNDO)!=0)
        Undo.reset();
    else {
        Undo.setup(Undo.FILTER, imp);
        ip.snapshot();
    }
    //ip.setMask(mask);
    ip.setCalibrationTable(cTable);
    ((PlugInFilter)theFilter).run(ip);
    if ((capabilities&PlugInFilter.SUPPORTS_MASKING)!=0)
        ip.reset(ip.getMask());  //restore image outside irregular roi
    IJ.showTime(imp, imp.getStartTime(), cmd + ": ", 1);
} else {
    Undo.reset(); // can't undo stack operations
    int n = stack.getSize();
    int currentSlice = imp.getCurrentSlice();
    Rectangle r = null;
    Roi roi = imp.getRoi();
    if (roi!=null && roi.getType()<Roi.LINE)
        r = roi.getBoundingRect();
    mask = imp.getMask();
    ip = imp.getProcessor();
    double minThreshold = ip.getMinThreshold();
    double maxThreshold = ip.getMaxThreshold();
    ip = stack.getProcessor(1);
    ip.setLineWidth(Line.getWidth());
    boolean doMasking = roi!=null && roi.getType()!=Roi.RECTANGLE
        && (capabilities&PlugInFilter.SUPPORTS_MASKING)!=0;
    if (minThreshold!=ImageProcessor.NO_THRESHOLD)
        ip.setThreshold(minThreshold,maxThreshold,ImageProcessor.NO_LUT_UPDATE);
    boolean doGarbageCollection = IJ.isWindows() && !IJ.isJava2();
    for (int i=1; i<=n; i++) {
        ip.setPixels(stack.getPixels(i));
        ip.setMask(mask);
        ip.setRoi(r);
        ip.setCalibrationTable(cTable);
        if (doMasking)
            ip.snapshot();
        ((PlugInFilter)theFilter).run(ip);
        if (doMasking)
            ip.reset(mask);
        if (doGarbageCollection && (i%10==0))
            System.gc();
        IJ.showProgress((double)i/n);
```

```
        }
        if (roi!=null)
            imp.setRoi(roi);
        IJ.showProgress(1.0);
        IJ.showTime(imp, imp.getStartTime(), cmd + ": ", n);
    }
    if ((capabilities&PlugInFilter.NO_CHANGES)==0) {
        imp.changes = true;
        //if (slices>1 && (type==ImagePlus.GRAY16||type==ImagePlus.GRAY32))
        //  imp.getProcessor().resetMinAndMax();
        imp.updateAndDraw();
    }
    ImageWindow win = imp.getWindow();
    if (win!=null)
        win.running = false;
    imp.unlock();
}


static Object runUserPlugIn(String commandName, String className, String arg,
                            boolean createNewLoader) {
    if (applet!=null)
        return null;
    String pluginsDir = Menus.getPlugInsPath();
    if (pluginsDir==null)
        return null;
    if (notVerified) {
        // check for duplicate classes in the plugins folder
        IJ.runPlugIn("ij.plugin.ClassChecker", "");
        notVerified = false;
    }
    PluginClassLoader loader;
    if (createNewLoader)
        loader = new PluginClassLoader(pluginsDir);
    else {
        if (classLoader==null)
            classLoader = new PluginClassLoader(pluginsDir);
        loader = classLoader;
    }
    Object thePlugIn = null;
    try {
        thePlugIn = (loader.loadClass(className)).newInstance();
        if (thePlugIn instanceof PlugIn)
            ((PlugIn)thePlugIn).run(arg);
        else if (thePlugIn instanceof PlugInFilter)
            runFilterPlugIn(thePlugIn, commandName, arg);
    }
    catch (ClassNotFoundException e) {
        if (className.indexOf('_')!=-1)
```

```
            IJ.error("Plugin not found: "+className);
        }
        catch (InstantiationException e) {IJ.error("Unable to load plugin (ins)");}
        catch (IllegalAccessException e) {IJ.error("Unable to load plugin (acc)");}
        return thePlugIn;
    }

...
```

ij.io.PluginClassLoader

```
package ij.io;
import java.io.*;
import java.net.*;
import java.util.*;

/** ImageJ uses this class loader to load plugins from the plugins
    folder. It's based on the FileClassLoader from
    "Java Class Libraries: Second Edition, Vol. 1"
    (http://java.sun.com/docs/books/chanlee/second_edition/vol1/).
*/
public class PluginClassLoader extends ClassLoader {
    String path;
    Hashtable cache = new Hashtable();
    public PluginClassLoader(String path) {
        this.path = path;
    }

    protected synchronized Class loadClass(String name, boolean resolve)
            throws ClassNotFoundException {
        Class c = (Class)cache.get(name); // try to find in cache
        //ij.IJ.log("loadClass: "+name+" ("+((c == null) ?
            "not in cache" : "in cache")+"), "+resolve);

        if (c == null) {  // Not in cache
            try {
                return findSystemClass(name);  // try system class loader
            } catch (ClassNotFoundException e) {}
            c = loadIt(path, name);    // Try to get it from plugins folder
            if (c==null)
                c = loadFromSubdirectory(path, name);  // Try to get it from subfolders
            if (c==null) {
                // try loading from ij.jar
                try {c = Class.forName(name);}
                catch (Exception e) {c=null;}
            }
            if (c==null)
                throw new ClassNotFoundException(name);
        }


        // Link class if asked to do so
        if (c != null && resolve)
            resolveClass(c);
        return c;
    }
```

```
    // Loads the bytes from file
    Class loadIt(String path, String classname) {
        String filename = classname.replace('.','/');
        filename += ".class";
        File fullname = new File(path, filename);
        //ij.IJ.write("loadIt: " + fullname);
        try { // read the byte codes
            InputStream is = new FileInputStream(fullname);
            int bufsize = (int)fullname.length();
            byte buf[] = new byte[bufsize];
            is.read(buf, 0, bufsize);
            is.close();
            Class c = defineClass(classname, buf, 0, buf.length);
            cache.put(classname, c);
            return c;
        } catch (Exception e) {
            return null;
        }
    }
}
```

Annexe 8

Echantillons de code pour le *pattern "plugin"*

Java

L'implémentation Java qui suit montre l'utilisation de plugins. Elle concerne un panneau d'affichage qui contient au maximum un plugin qui permet de visualiser un objet.

```
import javax.swing.JPanel;

public class ViewPanel extends JPanel {

        // abstract class for all view plug-ins
    public static abstract class ViewPlugIn
        extends JPanel implements PlugIn {
            // return true, if this panel can display o
        public abstract boolean canDisplay(Object o);
            // sets the object to be displayed
            // (it is assumed that canDisplay returned true for o)
        public abstract void display(Object o);
    }

        // the used plug-in loader
    private PlugInLoader loader;

        // constructs a new view Panel
        // (plugInType must be a subclass of ViewPlugIn)
    public ViewPanel(Class plugInType) {
            // test wether plugInType is a subclass of ViewPlugIn
        if (! ViewPlugIn.class.isAssignableFrom(plugInType))
            throw new IllegalArgumentException("wrong plugInType");

            // create a new plug-in loader for the given type
        loader = new PlugInLoader(plugInType);
    }
```

```
        // displays the given object, if possible
    public void display(Object o)
        throws PlugInNotFoundException {
            // first, the previous view panel is removed
        removeAll();

            // all plug-ins are retrieved
        PlugIn[] plugins = loader.getPlugIns();
            // ... and tested
        for (int i = 0; i < plugins.length; i++){
                // the type of all plug-ins is guaranteed to
                // be a subtype of ViewPlugIn (see constructor)
            ViewPlugIn plugin = (ViewPlugIn) plugins[i];
                // if a plug-in can display o, it is used
            if (plugin.canDisplay(o)){
                plugin.display(o);
                add(plugin);
                return;
            }
        }
            // in this case, not suitable plug-in has been found
        throw new PlugInNotFound("no such view plug-in");
    }
}
```

L'implémentation Java utilisée du `PlugInLoader` est basée sur la technique décrite en [LB01]. Après un appel à la méthode `display()`, tous les plug-ins du type spécifié sont recherchés. Après cela, ils sont interrogés, via la méthode "de vote" `canDisplay()`, sur leur capacité à afficher l'objet donné. Le premier plugin qui donne une réponse positive est choisi pour afficher l'objet. Remarquons encore que même si `ViewPanel` utilise un objet `PlugInLoader`, il peut aussi être vu comme un chargeur de plugins de plus haut niveau.

Perl

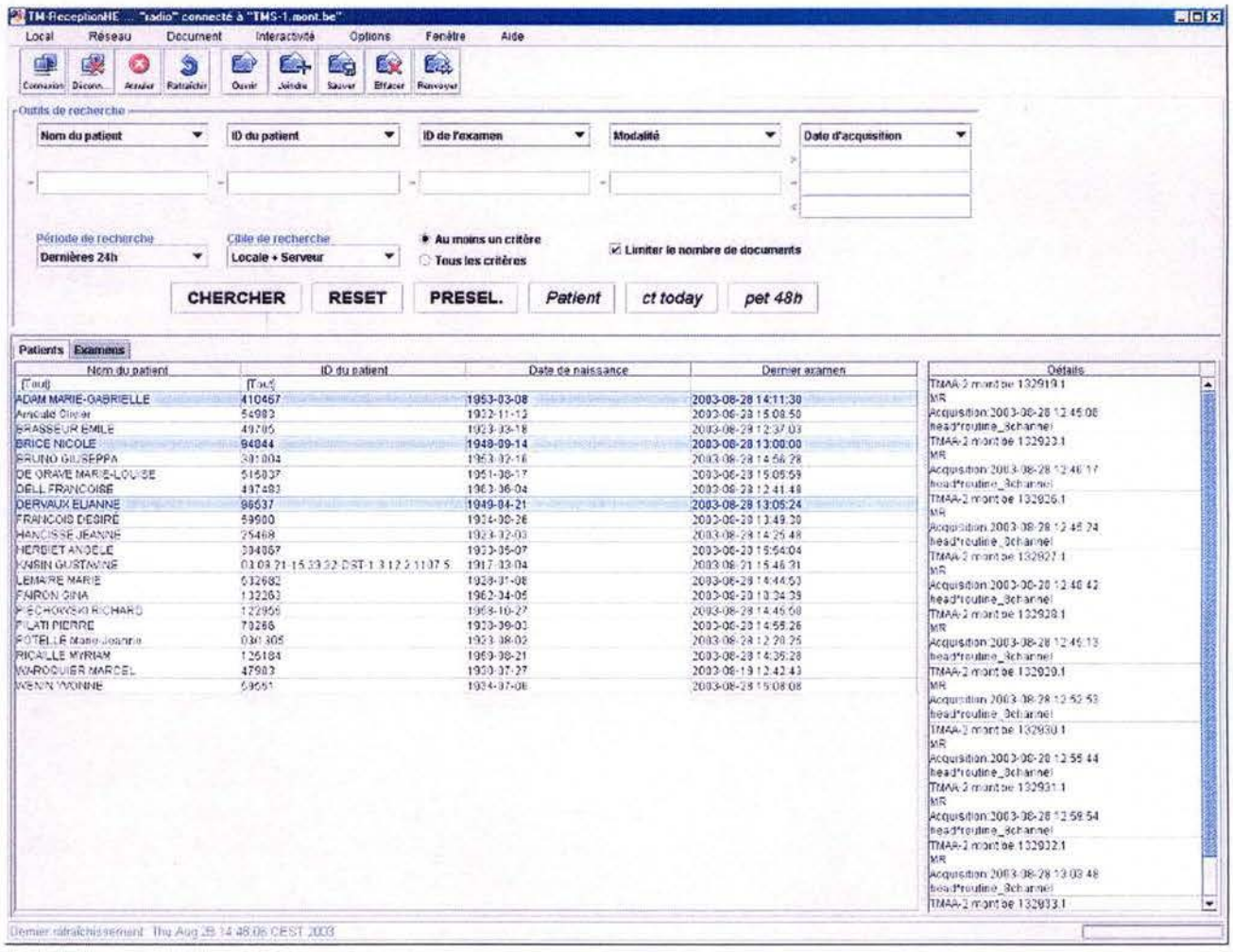
Il est possible d'implémenter une solution similaire en Perl. Le code qui suit présente une approche de base.

```
sub scan {
    my ($path) = @_;
    my $dir = new IO::Dir $path;
    die "Unable to scan $path: $!" unless defined $dir;
    my $filename; my %plugins = ();
    while (defined ($filename = $dir->read)){
        next unless $filename =~ /^(.*)\.pm$/;
        my $name = $1;
        my $module = "Plugins::$name";
        my $plugin;
        eval qq{
            use $module;
            \$plugin = new $module;
            die "Wrong Interface"
                unless (\$plugin->isa(\Plugins::Base\'));
        };
        if ($@) {warn "$module ignored: $@"; next; }
        $plugins{$name} = $plugin;
    }
    return \%plugins;
}
```

Un répertoire déterminé est scanné pour trouver d'éventuels plugins. Pour chaque module trouvé, on teste si celui-ci est chargeable dynamiquement et si il s'agit bien d'une implémentation de l'interface des plugins. La fonction `scan` retourne un pointeur vers un tableau contenant les plugins trouvés.

Annexe 9

Interface de Telemis pour la boîte de réception des images



Annexe 10

Comment écrire des plugins pour Telemis : manuel d'instruction

Introduction

Ce document décrit la démarche à suivre pour écrire correctement des plugins au niveau du viewer 2D du module " TM-Reception " de l'application Telemis. Il expose les conditions indispensables à respecter ainsi que quelques techniques conseillées lors de l'écriture d'un plugin. Enfin, il tente de définir un panel le plus complet possible des objets et méthodes susceptibles d'être utilisés par les personnes concevant des plugins. Il est important de remarquer que l'hypothèse de départ est que l'on cherche à écrire des outils supplémentaires permettant de travailler principalement au niveau de l'image de manière relativement indépendante vis-à-vis de l'application. Par conséquent on se concentrera d'avantage sur des objets et des méthodes permettant de récupérer un maximum d'informations sur les images affichées dans le viewer, afin de pouvoir les traiter de manière indépendante. Enfin on définira également les méthodes permettant de réinsérer dans le viewer le résultat du traitement de l'image. Nous terminerons par un petit mot sur l'interface de l'outil de développement de plugins.

Compilation des plugins

Le kit de développement de plugins permet de compiler les plugins à l'intérieur de l'application de sorte que l'on ne soit pas obligé de quitter cette dernière entre deux compilations successives d'un même plugin. En plus de la simple compilation des fichiers java, la présence d'un fichier de configuration dans le répertoire d'hébergement des plugins sera indispensable pour la bonne exécution de celui-ci. Ce fichier de configuration contient notamment le nom de tous les fichiers ".class" générés par une compilation. Par conséquent, afin d'éviter de devoir chipoter dans les répertoires d'installation de Telemis au début du développement du plugin, et de pouvoir construire le fichier de configuration automatiquement, l'outil de développement de plugins propose à l'utilisateur deux modes de compilation. Le premier est un mode de "construction de l'environnement" où le développeur travaille dans un environnement indépendant de Telemis et où la construction automatique du fichier de configuration peut se faire sans interférence avec d'éventuels fichiers de l'architecture Telemis. Le second est un mode final pour lequel la compilation insère les fichiers dans les répertoires de Telemis hébergeant les plugins, et met à jour l'interface de façon à rendre l'exécution possible à partir du viewer. Dans les deux cas, le développeur est averti des erreurs de compilation s'il y en a.

Cependant, le développeur devra respecter quelques contraintes afin que ces opérations

se déroulent correctement :

1. **Nom du fichier principal**

   Le nom du fichier java correspondant à la classe principale du plugin doit se terminer obligatoirement par "_". En effet, c'est ainsi que le système mis en place distingue les classes "plugins" des classes "librairies" dont se servent les plugins.

2. **Création et initialisation d'un fichier " xxx_.plugin"**

   Pour être installé soigneusement dans Telemis, un plugin doit toujours être accompagné d'un fichier du même nom que la classe principale, mais avec l'extension ".plugin". L'outil de construction de l'environnement crée automatiquement ce fichier et le remplit de façon adéquate. Toutefois, s'il advenait que la création soit douteuse ou que l'on désire modifier le nom du plugin dans l'interface Telemis, il est intéressant de savoir de quoi est composé ce fichier. La première ligne de ce fichier contient le nom sous lequel le plugin doit apparaître à l'intérieur de l'application Telemis. C'est ce nom qui figurera sur l'item du menu d'exécution. A partir de la deuxième ligne, et chaque fois sur des lignes différentes, doivent figurer les noms de tous les fichiers utilisés par le plugin à l'exécution. On y trouve donc les éventuels fichiers de librairies ".jar" et ".zip" que le système ajoutera automatiquement au *classpath*. On y met également tous les fichiers ".class" utilisés par le plugin, ainsi que leurs descendants[1]. C'est la complétude de ce fichier ".plugin" qui garantit un bon nettoyage du répertoire de Telemis dans lequel se situent les plugins, au moment de l'effacement. Par conséquent, il faudra s'assurer que toute modification du contenu de ce fichier conserve le respect de cette contrainte. La démarche à suivre pour la construction automatique de ce fichier est expliquée à la fin de ce document dans la section "L'interface de l'outil de développement".

   Exemple : Un plugin pour envoyer par e-mail certaines images présentes sur le viewer a déjà été conçu. Le fichier principal à compiler s'appelle *Send_in_mail_.java* et il devra utiliser le résultat de la compilation du fichier *MailSendChooser.java* qui utilise lui même des classes contenu dans *smtp.jar*. Le résultat de la compilation donne les fichiers *Send_in_mail_.class*, *MailSendChooser.class*, *ListAdressPane.class* et *ListAdressPane$1.class*. On veut que l'item de menu correspondant à l'exécution de ce plugin soit libellé "Send images via e-mail". L'édition du fichier *Send_in_mail_.plugin* présent dans le répertoire devra forcément donner :

[1] par "descendants" on entend tous les fichiers générés par la compilation d'un fichier java. Par exemple, un des plugins déja conçus utilise un fichier *MailSendChooser.java* qui, à la compilation, génère les fichiers *MailSendChooser.class*, *ListAdressPane.class* et *ListAdressPane$1.class*. Ces trois derniers sont les descendants du fichier *MailSendChooser.java*

```
Send images via e-mail
Send_in_mail_.class
MailSendChooser.class
ListAdressPane.class
ListAdressPane$1.class
smtp.jar
```

Remarque : Par défaut, l'outil "création de l'environnement" met comme nom de plugin (la première ligne du fichier) le nom du fichier ".java" principal sans l'extension ("Send_in_mail_" dans ce cas) .

3. **Emplacement des fichiers nécessaires**

Tous les fichiers nécessaires à la compilation d'un plugin doivent se trouver dans le même répertoire. L'outil de compilation permettra de naviguer sur le disque afin de sélectionner ce répertoire.

Exemple : les fichiers *Send_in_mail_.java*, *MailSendChooser.java*, *smtp.jar* et *Send_in_mail_.plugin* de l'exemple de la section précédente doivent se situer dans le même répertoire au moment de la compilation.

4. **Code obligatoire et technique conseillée**

Certaines parties de code sont obligatoires pour que le plugin soit reconnu par l'outil de développement et pour pouvoir se servir des classes Telemis relatives au viewer. Dans les "imports" en début de code il faudra donc ajouter ceci :

import telemis.om.image.;* (afin d'utiliser les objets relatifs au viewer)
import telemis.plugins.;* (pour pouvoir accéder à l'interface Plugin et à d'autres outils créés)

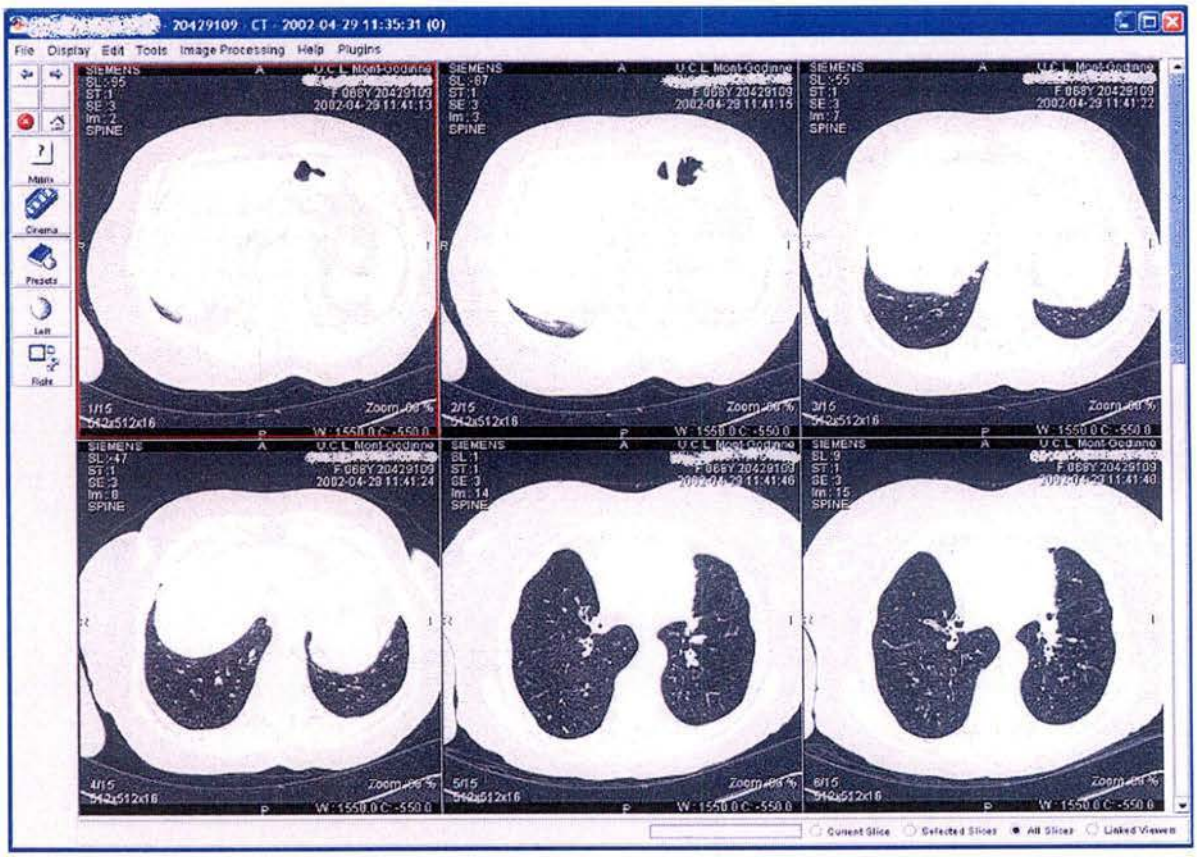
De même, tout plugin doit implémenter l'interface *telemis.plugins.Plugin* et contenir la méthode *runit* dont la signature est *public void runit(Object iw)*. C'est cette méthode qui est appelée lors d'une pression sur l'item de menu du plugin et qui contient le code à exécuter. On passera en argument à cette méthode l'instance de *ImageWindow* à partir de laquelle le plugin est appelé de façon à pouvoir travailler avec celle-ci. Un type-casting vers *ImageWindow* sera donc nécessaire à l'intérieur de la méthode *runit*. Afin que le plugin s'exécute comme un processus indépendant et ne s'accapare pas trop de temps CPU, il est conseillé d'en faire un thread indépendant. On peut alors utiliser une structure semblable à celle-ci dans le fichier source du plugin :

Fichier XXX_.java

```
import telemis.om.image.*;
import telemis.plugins.*;

...

public class XXX_ implements Plugin, Runnable{

...

ImageWindow myImageWindow = null;

...

public void runit (Object iw){
        myImageWindow = (ImageWindow)iw;
        Thread plugin = new Thread(this);
        plugin.start();
}

public void run( ){
        //code à exécuter
}

...

}
```

Panel d'objets et méthodes Telemis

Présentation de la structure du viewer Telemis



Voici le viewer 2D du module Telemis-Réception, à partir duquel il est possible de récupérer des informations sur les images qu'il contient, et d'insérer de nouvelles images après traitement. Ce viewer est implémenté par la classe *telemis.om.image.ImageWindow* et est notamment une extension de *javax.swing.JFrame*. Il contient un certain nombre de *telemis.om.image.ImageSlice*, qui sont les objets responsables de l'affichage de *telemis.om.image.dicomImage*. Le cadre rouge sur l'image du viewer délimite une *ImageSlice*. Il est très important de ne pas confondre une " *dicomImage* " Telemis et une image Dicom du standard du même nom. En effet, une image Dicom peut contenir plusieurs tranches et former une image volumique, alors qu'une instance de *dicomImage* correspond en fait au tableau de pixel (et autres informations relatives) d'une seule tranche d'une image " complète ". Par conséquent, à une instance de *ImageSlice* est associée une et une seule instance de *dicomImage*. Chaque instance de *ImageWindow* et *ImageSlice* trouve respectivement ses caractéristiques dans une instance de *telemis.om.image.ImageWindowParam* et *telemis.om.image.SliceParam*. Par exemple *SliceParam* donne des informations sur les facteurs de zoom, les éventuels off-

sets, le contraste et la luminosité ou encore sur l'éventuelle sélection par l'utilisateur de la slice concernée. Il a été choisi, pour une question de généricité principalement, de ne pas trop développer les explications sur ces deux dernières classes, au profit de la classe *dicomImage* et de ses classes relatives, qui sont plus proches des données brutes. En effet, on peut considérer que le rendu visible dans le viewer est le résultat de l'application de paramètres de transformation (notamment les *SliceParam*) sur les données brutes (principalement dans *dicomImage*). Or, ce paramétrage est une chose propre à chaque logiciel tandis que la façon de représenter les données brutes est bien souvent identique au travers des programmes d'imagerie. Afin de ne pas devoir rentrer trop en profondeur dans le paramétrage et de pouvoir appliquer des algorithmes génériques, il peut être préférable de se concentrer sur ce qui est le plus proche des " *raw datas* ", c'est-à-dire les tableaux de pixels stockés dans les *dicomImage*. Enfin, à chaque *ImageWindow* est associé un *telemis.om.image.dicomStudy*, dans lequel on retrouve principalement des informations sur l'ensemble des *dicomImage* affichées dans le viewer.

Répertoire des méthodes et objets intéressants

Voici un répertoire d'objets et méthodes des classes citées à la section précédente susceptibles d'être utilisés dans le code des plugins :

ImageWindow

- *public dicomStudy getDicomStudy()*
  Renvoie l'instance de dicomStudy associée à l'*ImageWindow*.

- *public void repaint()*
  C'est le " repaint " classique à appeler après modification d'un composant du viewer. Ce n'est toutefois pas cette méthode que l'on utilisera pour afficher une modification faite au niveau du tableau de pixels dans *dicomImage*.

- *public void addACommand(int theCommand)*
  Passée avec l'argument *ImageWindow.REDRAW*, c'est la méthode à appeler pour redessiner l'*ImageWindow*, notamment après modification d'un ou plusieurs tableaux de pixels dans les *dicomImage*.

- *public void setLastClickedImageSlice(ImageSlice is)*
  Cette méthode permet de donner explicitement la dernière slice sur laquelle on a cliqué. Voir plugin "HtmlInfo".

- *public ImageSlice getLastClickedImageSlice()*

Cette méthode permet de récupérer la dernière slice sur laquelle on a cliqué. Elle peut notamment être intéressante dans un plugin où l'on désire récupérer la slice sur laquelle on est en train de travailler. Voir plugin "HtmlInfo".

- *public ImageSlice[] myImageSlices*
  Il s'agit du champ qui contient les slices de l'*ImageWindow*. La première slice (encadrée en rouge sur l'image) est l'élément *myImageSlices[0]*.

dicomStudy

- *public Vector data*
  Ce vecteur contient les *dicomImage* affichées dans les slices du viewer. Par conséquent, c'est par ce champ que l'on récupèrera les données brutes. Ex : si nous disposons dans un de nos plugins du champ *ImageWindow myImageWindow*, alors *myImageWindow.getDicomStudy().data.elementAt(0)* nous renverra la *dicomImage* affichée par la première slice du viewer.

- *public int getNbSlices()*
  Méthode renvoyant le nombre de slices contenues dans le document affiché dans le viewer.

- *public boolean isSelected(int i)*
  Cette méthode permet de savoir si une slice en question est l'objet d'une sélection de la part de l'utilisateur.

- *public void selectAll()*
  Il s'agit de la méthode à appeler lorsque l'on veut explicitement sélectionner toutes les slices du document.

dicomImage

Comme dit auparavant, c'est cette classe qui héberge le tableau de pixels de la slice qui lui est associée. Dans Telemis, les pixels peuvent être codés de trois façons différentes, soit sur 8, 16 ou 24 bits. Nous aurons donc respectivement la possibilité d'avoir des tableaux de *byte*, de *short* ou d'*int* (puisqu'il n'existe pas en java de type numérique codé sur 24 bits).

Remarque : Il est très rare à ce jour de trouver des images Dicom codées sur 8 ou 24 bits

et il en est donc de même pour les documents Telemis. Par conséquent, il n'est pas toujours nécessaire de traiter dans les plugins le cas des images 8 ou 24 bits.

- *public int getDepth()*
  Renvoie le nombre de bits sur lequel l'image est codée, c'est-à-dire soit 8, 16 ou 24.

- *public int getWidth()*
  Renvoie la largeur de l'image en nombre de pixels.

- *public int getHeight()*
  Renvoie la hauteur de l'image en nombre de pixels.

- *public int getMax()*
  Retourne la valeur du pixel ayant la plus grande intensité (la plus grande valeur).

- *public int getMin()*
  Retourne la valeur du pixel ayant la plus faible intensité (la plus petite valeur). Cette valeur est dans la grande majorité des cas égale à 0.

- *public float getPixelSpacingX()*
  Renvoie la taille en mm d'un pixel, sur l'axe horizontal.

- *public float getPixelSpacingY()*
  Renvoie la taille en mm d'un pixel, sur l'axe vertical.

Remarque : les pixels représentant presque toujours des carrés, les valeurs renvoyées par *getPixelSpacingX()* et *getPixelSpacingY()* sont quasi toujours identiques.

Les six méthodes suivantes sont les méthodes d'accès aux pixels, les plus importantes dans notre cas. Afin de savoir laquelle consulter, il peut être bon de récupérer dans un premier temps la "profondeur" du pixel de l'image grâce à la méthode *getDepth* (voir ci-dessus).

- *public byte[] getPixelsByte()*
  Retourne le tableau de pixels dans le cas d'une "profondeur" de 8.

- *public short[] getPixelsShort()*
  Retourne le tableau de pixels dans le cas d'une "profondeur" de 16. C'est de loin la plus utilisée des méthodes d'accès en lecture (voir remarque précédente).

- *public int[] getPixelsInt()*

Retourne le tableau de pixels dans le cas d'une " profondeur " de 24.

- *public boolean setPixelsByte(byte[] pixe)*
  Permet d'insérer un tableau de pixels dans le cas d'une "profondeur" de 8.

- *public boolean setPixelsShort(short [] pixe)*
  Permet d'insérer un tableau de pixels dans le cas d'une "profondeur" de 16. C'est de loin la plus utilisée des méthodes d'accès en écriture (voir remarque précédente).

- *public boolean setPixelsInt(int [] pixe)*
  Permet d'insérer un tableau de pixels dans le cas d'une "profondeur" de 24.

ImageSlice

- *public dicomImage getDicomImage()*
  Cette méthode permet de récupérer la dicomImage affichée par la slice.

- *public ImageWindow getImageWindow()*
  Cette méthode permet de récupérer l'ImageWindow dans laquelle se trouve la slice.

Quelques fragments de code intéressants

Au cours du développement de certains plugins, il s'est avéré que quelques fonctions particulières (c'est-à-dire sans rapport direct avec l'objectif précisé dans l'introduction) pouvaient être intéressantes à utiliser. Deux de celles-ci sont explicitées ci-après.

Réaction au click sur le troisième bouton de la souris

Ce code montre comment il est possible d'exécuter une action suite à un click sur le bouton "central" de la souris. Dans ce cas, il s'agit de récupérer la slice sur laquelle était positionné le curseur au moment de la pression. Dans le code qui suit, l'action se termine au moment du relâchement du bouton de souris.

Dans un premier temps il faut ajouter des écouteurs d'événements de souris au niveau de chaque slice :

```
for (int i=0; i<nbslice ; i++){
    ((myImageWindow.myImageSlices)[i]).addMouseListener(new TheMouseListener());
}
```

Il faut bien sûr définir la classe qui "écoute"...

```
class TheMouseListener extends java.awt.event.MouseAdapter{

    public void mousePressed(java.awt.event.MouseEvent e){
        press(e);
    }

    public void press(java.awt.event.MouseEvent e){
        myImageWindow.setLastClickedImageSlice((ImageSlice)(e.getComponent()));
        lastMousePressedPosition=e.getPoint();
        lastClickModifiers=e.getModifiers();
        if ((lastClickModifiers & java.awt.event.InputEvent.BUTTON2_MASK)!=0) {
            //On place ici le code à effectuer en cas de pression sur le bouton
            lastDragPosition=e.getPoint();
            lastMousePosition=e.getPoint();
            //Dans ce cas-ci, une instance de ImageMenu (défini par un développeur
            //de plugins) est créée avec pour paramètres notamment des informations
            //sur la dernière slice cliquée" et sur la position du curseur.
        myImageMenu = newImageMenu(myImageWindow.getLastClickedImageSlice().getDicomImage().number,
        myImageWindow.getDicomStudy(), myImageWindow.getLastClickedImageSlice().getLocationOnScreen());
        myImageMenu.showMenu(myImageWindow.getLastClickedImageSlice(),e.getX(),e.getY());
        }
    }

    public void mouseReleased(java.awt.event.MouseEvent e) {
        if ((lastClickModifiers & java.awt.event.InputEvent.BUTTON2_MASK)!=0){
            //On place ici le code à effectuer en cas de relâchement du bouton.
            myImageMenu.executeMenu();
            myImageMenu.hideMenu();
        }
    }
}
```

Création d'un document Telemis "de base" à afficher dans un nouveau viewer

Il peut être intéressant lorsqu'on dispose d'un ou plusieurs tableaux de pixels constituant une image, de pouvoir afficher cette dernière dans un nouveau viewer. Cela nécessite la création d'un nouveau document Telemis et l'instanciation de quelques classes.
Voici comment procéder :

```
//Création d'un dicomStudy
dicomStudy toBuild=new dicomStudy();
toBuild.setModality(new String("SC"));
String newIdentifier = toBuild.getId()+"."+(counter++);
dm newDocument=generateNewDocument(newIdentifier);
toBuild.setId("image."+newDocument.getId()+"."+(counter++));
//on place en argument de la méthode suivante le nom que l'on veut donner au document
newDocument.setName("nom document");
createTheStudy(toBuild);
toBuild.setDmRef(newDocument);
newDocument.setState(newDocument.NORMAL);
ViewerURL v=new ViewerURL(thisTelemisApplication.getName(),newIdentifier,
    toBuild.getId(),new String(""+dicomStudy.getIncrement()));
ImageWindowParam iwp=DisplayManager.findAndApplyDisplay(toBuild);
ImageWindowdw=new ImageWindow(iwp,toBuild);
dw.setURL(v);
dw.setName("Test Image Import");
dw.show();
dw.setDecompressionFinished();


public dm generateNewDocument(String newId){
    dm newdocument=new dm();
    newdocument.setId(newId);
    try {
        newdocument.setCreator(thisTelemisApplication.getProfile().getName());
        }
    catch (Exception e) {}
    return newdocument;
}


public boolean createTheStudy(dicomStudy ds) {
    return createOneImageStudy(ds);
}


public boolean createOneImageStudy(dicomStudy ds) {
    //axial chosen
    int counter=0;
    dicomImage di=null;
    for (int i=0 ; i<nbimages ; i++) {
        di = makeImage(i);
        di.number=counter;
        di.getSliceParam().sliceNumber=counter;
        ds.data.addElement(di);
        counter++;
    }
```

```
    ds.setNbSlices(counter);
    ds.computeHW();
}

public dicomImage makeImage(int nb) {
    dicomImage yz = null;
    //on a considéré ici que le cas des images 16 bits
    if (depth == 16){
        //c'est à cet endroit que l'on remplit le tableau de pixels
        yz = new dicomImage(width,height,16);
        short[] pixtab = new short[height*width];
        int cpt = height*width*nb;
        for (int d =0 ; d<pixtab.length; d++,cpt++ ){
            pixtab[d] = shortpix[cpt];
        }
        yz.setPixelsShort(pixtab);
    }
    yz.setState(0);
    yz.forceInitializeParam();
    return yz;
}
```

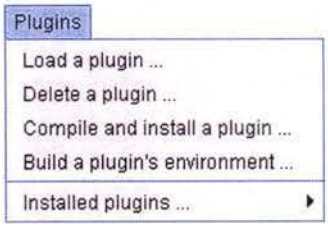
On aura pris soin de définir les imports suivants :

```
import telemis.awt.*;
import telemis.om.image.*;
import telemis.om.*;
import telemis.generic.*;
import telemis.net.*;
import telemis.util.*;
```

Remarque : il aurait été possible de donner plus d'informations pour préciser l'état du document (nom patient, heure , etc...) mais le but est ici simplement de pouvoir afficher une image "basique" dans le viewer.

L'interface de l'outil de développement

Lorsque l'outil de développement de plugins est installé, un nouveau menu " Plugins " est ajouté au viewer 2D. Le menu se présente comme ceci :



- **Load a plugin ...**

  A partir de cet item, on peut charger un plugin déjà compilé. La copie des fichiers nécessaires ainsi que l'ajout de l'item pour exécuter le plugin se feront automatiquement. Une boîte de sélection de fichier permettra de sélectionner sur le disque un fichier ".plugin" à partir duquel le plugin va se charger. Il faudra cependant veiller à ce que tous les fichiers nécessaires à l'exécution du plugin se situe dans le même répertoire, c'est-à-dire la classe principale et ses descendants, les fichiers répertoriés dans le ".plugin" et le ".plugin". Cette fonction peut en fait être directement utilisée après avoir construit l'environnement dans un répertoire quelconque du disque, grâce à l'outil " Build a plugin's environment ".

- **Delete a plugin ...**

  Cet item lance la désinstallation d'un plugin après sélection de celui-ci dans le répertoire d'hébergement. Une boîte de sélection de fichier permettra de sélectionner le ".plugin" concerné. Une fois le plugin choisi, l'effacement des fichiers inutiles et de l'item correspondant seront automatiques.

- **Compile and install a plugin ...**

  Cette fonction permet la compilation d'un plugin (à partir d'un ".java" sélectionné grâce à une boîte de sélection) et son installation dans Telemis. Un éventuel rapport d'erreurs de compilation sera affiché. On veillera à respecter les obligations décrites dans la section "Compilation des plugins".

- **Build a plugin's environment ...**

  Cette fonction permet la compilation d'un plugin (à partir d'un " .java " sélectionné grâce à une boîte de sélection) dans un répertoire quelconque indépendant de Telemis. Elle permet ainsi de déboguer le code du plugin sans interférer avec Telemis. Un éventuel rapport d'erreurs de compilation sera affiché. On veillera à respecter les obligations

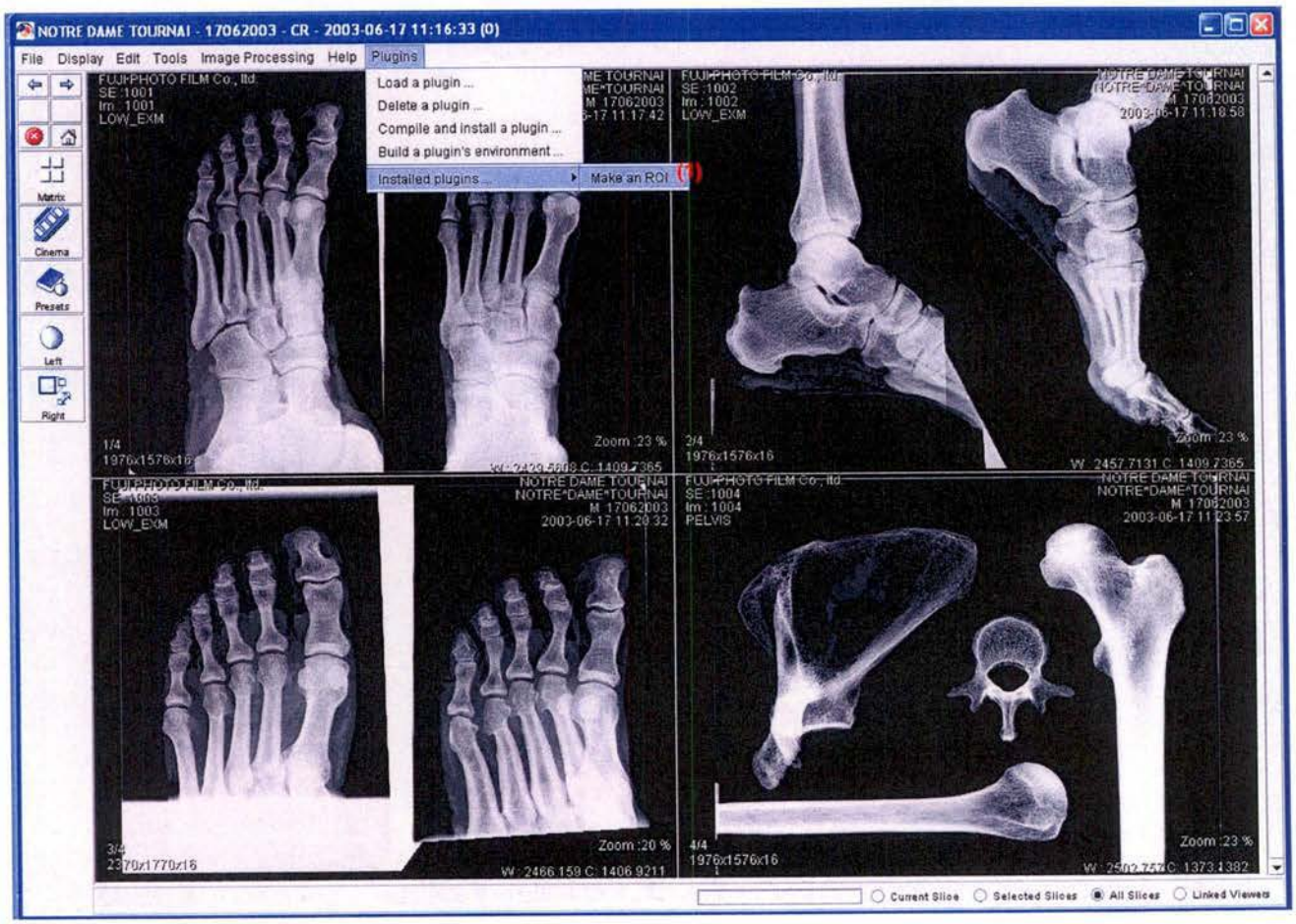
décrites dans la section "Compilation des plugins". Notons que cet outil génère automatiquement le fichier ".plugin" et le remplit correctement. Ce dernier est placé dans le même repertoire que le résultat de la compilation.

Une fois un plugin installé, l'item permettant de l'exécuter s'ajoute à l'interface, dans le sous-menu " Installed plugins ".

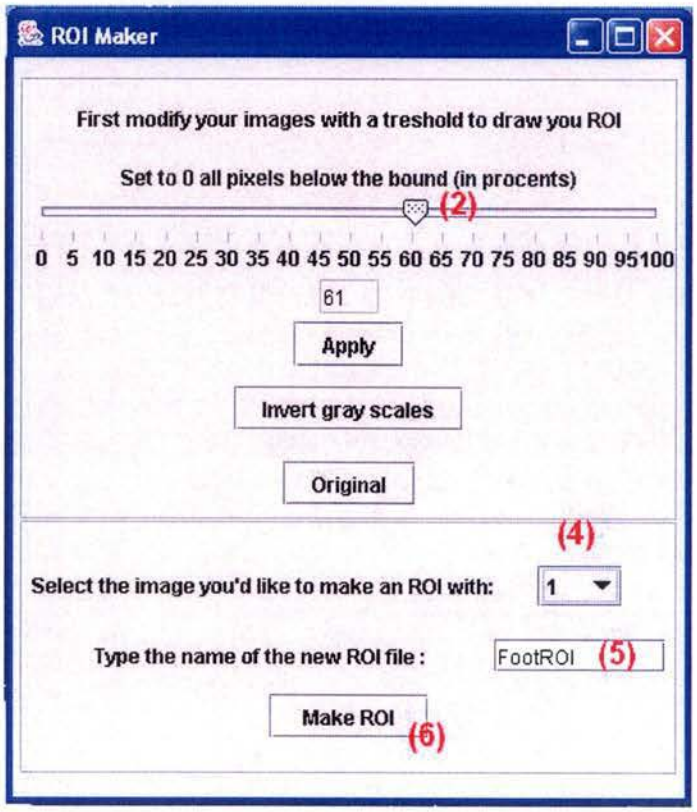
Annexe 11

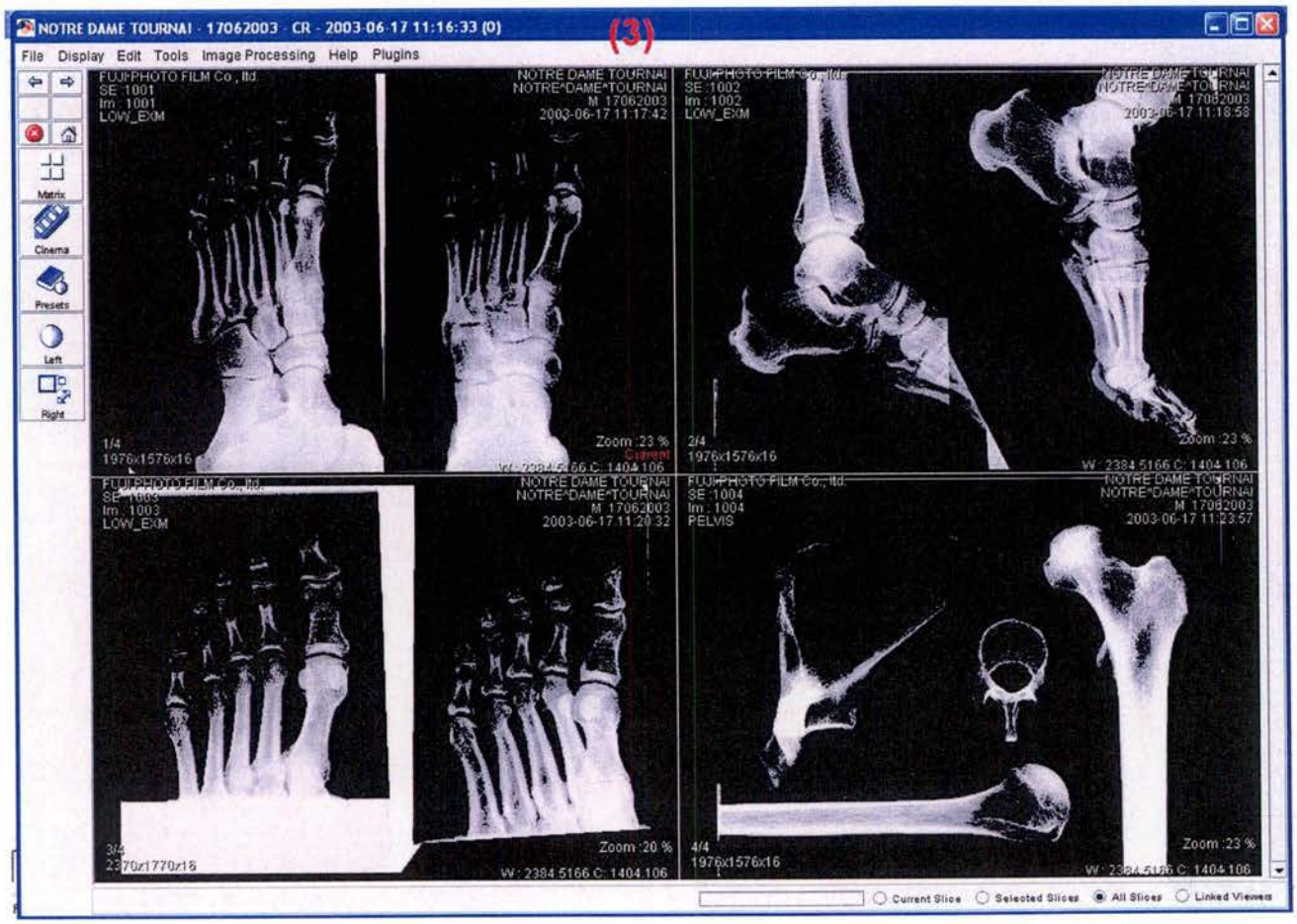
Exemple d'utilisation du plugin "*ROI Maker*"

Pour lancer l'exécution du plugin, il faut cliquer sur l'item qui lui correspond dans le menu des plugins installés. Celui-ci est pointé par le symbole (1) sur la photo.



L'interface du plugin illustrée par la photo suivante apparaît alors. Ensuite, l'utilisateur exprime en (2) la quantité des pixels qu'il veut noircir. Pour ce faire, il sélectionne une limite, calculée en pourcents par rapport à l'échelle de valeur des pixels, en dessous de laquelle il faut noircir les pixels. Lorsqu'il presse le bouton "Apply", les modifications apparaissent directement sur les images dans le viewer. On peut les voir sur la photo à la page suivante. Il s'agit de l'opération (3). Lorsque l'utilisateur a fait ressortir à sa guise les régions qu'il veut mettre en évidence, il doit alors choisir en (4) l'image qu'il voudra capturer pour en faire une région d'intérêt. En (5), il donne explicitement le nom du fichier qui contiendra cette région. Il lui reste alors à presser le bouton "Make ROI", en (6), pour enregistrer la région d'intérêt, qui sera constituée de l'ensemble des points qui n'ont pas été noircis auparavant. Le fichier résultat se place dans un répertoire déterminé qui contient l'ensemble des fichiers créés par ce plugin. Lors de l'utilisation du plugin "Import ROI", il suffira de sélectionner un de ces fichiers pour que la région d'intérêt qu'il contient se superpose à une image contenue dans le viewer, sélectionnée précédemment.




NOTRE DAME TOURNAI - 17062003 - CR - 2003-06-17 11:16:33 (0)
(3)
File Display Edit Tools Image Processing Help Plugins
Matrix
Cinema
Presets
Left
Right
FUJIPHOTO FILM Co., ltd.
SE :1001
Im : 1001
LOW_EXM
NOTRE DAME TOURNAI
NOTRE^DAME^TOURNAI
M 17062003
2003-06-17 11:17:42
1/4
1976x1576x16
Zoom :23 %
Current
W : 2384 5166 C: 1404 106
FUJIPHOTO FILM Co., ltd.
SE :1002
Im : 1002
LOW_EXM
NOTRE DAME TOURNAI
NOTRE^DAME^TOURNAI
M 17062003
2003-06-17 11:18:58
2/4
1976x1576x16
Zoom :23 %
W : 2384 5166 C: 1404 106
FUJIPHOTO FILM Co., ltd.
SE :1003
Im : 1003
LOW_EXM
NOTRE DAME TOURNAI
NOTRE^DAME^TOURNAI
M 17062003
2003-06-17 11:20:32
3/4
2370x1770x16
Zoom :20 %
W : 2384 5166 C: 1404 106
FUJIPHOTO FILM Co., ltd.
SE :1004
Im : 1004
PELVIS
NOTRE DAME TOURNAI
NOTRE^DAME^TOURNAI
M 17062003
2003-06-17 11:23:57
4/4
1976x1576x16
Zoom :23 %
W : 2384 5166 C: 1404 106
Current Slice
Selected Slices
All Slices
Linked Viewers

Annexe 12

Code-source du plugin *Import ROI*

```
import telemis.om.image.*;
import telemis.plugins.*;
import telemis.util.*;

import java.util.Vector;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFileChooser;


import java.io.File;
import java.io.FileInputStream;
import java.io.DataInputStream;

public class Import_ROI_ implements Plugin, Runnable {

    ImageWindow myImageWindow = null;
    String dir = "C:\\Bruno\\rois";
    File file = null;
    dicomStudy ds = null;
    Vector data = null;
    private int theHeight;
    private int theWidth;
    private float theSizeX;
    private float theSizeY;
    int[] xtab = new int[200000];
    int[] ytab = new int[200000];
    int numberOfImages = 0;
    javax.swing.JComboBox liste = new javax.swing.JComboBox();
    FileInputStream in = null;
    DataInputStream instream = null;


    public void runit (Object iw){
        myImageWindow = (ImageWindow)iw;
        Thread plug = new Thread(this);
        plug.start();
    }

    public void run(){
        ds = myImageWindow.getDicomStudy();
        data = ds.data;
        numberOfImages = data.size();
```

```
JFileChooser chooser = new JFileChooser();
chooser.setCurrentDirectory(new File(dir));
int res = chooser.showOpenDialog(null);
String pathin = "";
if (res == JFileChooser.APPROVE_OPTION) {
        pathin = chooser.getSelectedFile().getPath();
}
file = new File(pathin);
//byte[] bytetab = new byte[4];
try{
    in = new FileInputStream(file);
    instream = new DataInputStream(in);
    //instream.read(bytetab);
    //lu = bytetab_to_int(bytetab);
    theHeight = instream.readInt();
    //instream.read(bytetab);
    //lu = bytetab_to_int(bytetab);
    theWidth = instream.readInt();
    theSizeX = instream.readFloat();
    theSizeY = instream.readFloat();
    System.out.println(theSizeX);
    System.out.println(theSizeY);
    fillTabs();
    /*int x;
    int y;
    int divent;
    while (instream.available() >= 4){
        //lu = bytetab_to_int(bytetab);
        lu = instream.readInt();
        x = (lu % theWidth)+1;
        xtab[i] = x;
        divent = lu / theWidth;
        y = divent + 1;
        ytab[i] = y;
        i++;
    }*/
    instream.close();
}
catch (Exception e){e.printStackTrace();}
fillList(liste);

javax.swing.JLabel label = new javax.swing.JLabel
        ("Select on which image you want to paste the ROI : ");
javax.swing.JButton ok = new javax.swing.JButton("Ok");
ok.addActionListener (new ActionListener () {
      public void actionPerformed (ActionEvent evt) {
          okActionPerformed (evt);
      }
  }
```

```
      );
    javax.swing.JFrame frame = new javax.swing.JFrame("Image selection");
    javax.swing.JPanel panel = new javax.swing.JPanel();
    frame.setDefaultCloseOperation(javax.swing.JFrame.DISPOSE_ON_CLOSE);
    frame.setSize(300,100);
    java.awt.Container contentpane = frame.getContentPane();
    contentpane.add(panel);
    panel.add(label);
    panel.add(liste);
    panel.add(ok);
    frame.show();
}

public void okActionPerformed (ActionEvent evt){
    int choix = ( ((Integer)(liste.getSelectedItem())).intValue() ) -1;
    dicomImage di = (dicomImage)data.elementAt(choix);
    ObjectROIBP overlay = new ObjectROIBP(ds,di,xtab,ytab);
    myImageWindow.addACommand(myImageWindow.REDRAW);
}

private void fillList(javax.swing.JComboBox liste){
    for (int i=1; i <= numberOfImages; i++ ){
        liste.addItem(new Integer(i));
    }
}

private void fillTabs(){
    //2 cas possibles : (1)image de depart de meme format que l'originale
                        (2)formats différents

    int x;
    int y;
    int lu;
    int i = 0;
    int divent;

    //les variables propres àl'image sur laquelle on doit insérer la ROI
    dicomImage di = (dicomImage)data.elementAt(0);
    int ht = di.getHeight();
    int wt = di.getWidth();
    float sx = di.getPixelSpacingX();
    float sy = di.getPixelSpacingY();
    System.out.println(sx);
    System.out.println(sy);
    //(1)
    if ((ht==theHeight)&&(wt==theWidth)&&(sx==theSizeX)&&(sy==theSizeY)){
        try{
            while (instream.available() >= 4){
                lu = instream.readInt();
```

```
                x = (lu % theWidth)+1;
                xtab[i] = x;
                divent = lu / theWidth;
                y = divent + 1;
                ytab[i] = y;
                i++;
            }
        }
        catch (Exception e){}
    }
    //(2)
    else{
        float xbis;
        float ybis;
        Float temp;
        try{
            while (instream.available() >= 4){
                lu = instream.readInt();
                x = (lu % theWidth)+1;
                xbis = ((x-1)*theSizeX)/sx;
                temp = new Float(xbis);
                x = temp.intValue();
                //xtab[i] = x;
                divent = lu / theWidth;
                y = divent + 1;
                ybis = ((y-1)*theSizeY)/sy;
                temp = new Float(ybis);
                y = temp.intValue();
                //ytab[i] = y;
                if ((x<=wt)&&(y<=ht)){
                    if ((i>0)&&((xtab[i-1]!=x)||(ytab[i-1]!=y))){
                        xtab[i] = x;
                        ytab[i] = y;
                        i++;
                    }
                    else if (i==0){
                        xtab[i] = x;
                        ytab[i] = y;
                        i++;
                    }
                }
            }
        }
        catch (Exception e){}
    }

}

/*private static int bytetab_to_int(byte[] bytetab){
```

```
        int int1 = (int)bytetab[0];
        int int2 = (int)bytetab[1];
        int int3 = (int)bytetab[2];
        int int4 = (int)bytetab[3];
        int1 = (int1 & 0xff) << 24;
        int2 = (int2 & 0xff)<< 16;
        int3 = (int3 & 0xff) << 8;
        int4 = int4 & 0xff;
        int res = int1 + int2 + int3 + int4;
        return(res);

    }*/


}
```

Annexe 13

Code-source des classes essentielles au système de plugins de Telemis

Interface Plugin

```
package telemis.plugins;


public interface Plugin {
    public void runit(Object iw);
}
```

Classe PluginClassLoader

```
package telemis.plugins;

import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.util.*;

public class PluginClassLoader extends ClassLoader {

    String path;

    public PluginClassLoader(String path) {
        this.path = path;
    }

    public PluginClassLoader() {
    }

}
```

Classe Compiler

```
package telemis.plugins;

import telemis.om.image.ImageWindow;
import telemis.util.*;
import java.awt.*;
import java.io.*;
import java.io.ByteArrayOutputStream;
import javax.swing.JFileChooser;

public class Compiler implements Runnable{

    private static sun.tools.javac.Main javac;
    private static ByteArrayOutputStream output;
    private static String dir,name,workDir;
    private static String pluginspath;
    private MyPluginExtension myPluginExtension = null;
    String newInClasspath = "";

    public Compiler(MyPluginExtension plext,String path){
        myPluginExtension = plext;
        pluginspath = path;
    }

    public void run(){
        compilePlus();
    }

    boolean isJavac() {
        try {
            if (javac==null) {
                output = new ByteArrayOutputStream(4096);
                javac = new sun.tools.javac.Main(output, "javac");
            }
        } catch (NoClassDefFoundError e) {
            new Error_Editor("Missing javac","This JVM does not include
                the javac compiler. Javac is\n"
                +"included with the Windows and Linux versions of ImageJ.\n"
                +"Mac OS 9 users must install Apple's Java SDK.");
            return false;
        }
        return true;
    }

    // open the .java source file
     boolean open(String msg) {
        boolean okay;
```

```
    String name=null;
    String dir=null;
    JFileChooser choixFichier;
    ExtensionFilter filtre;
    int res;
    choixFichier = new JFileChooser();
    filtre = new ExtensionFilter();
        filtre.addExtension("_.java");
        filtre.setDescription(".java");
        choixFichier.setDialogTitle("Compile a plugin");
        choixFichier.setFileFilter(filtre);
        res = choixFichier.showDialog(null," Compile");
    if (res == JFileChooser.APPROVE_OPTION) {
            File selec = choixFichier.getSelectedFile();
            name = choixFichier.getName(selec);
            String pluginName = name.substring(0,(name.length()-4));
            pluginName = pluginName+"plugin";
            if (myPluginExtension.pluginExists(pluginName)){
                TelemisDialog.dialog("Plugin already installed",
                "This plugin is already installed.
                    Please delete it before recompiling.",
                Language.translate("OK"),
                Language.translate("Cancel"), null, TelemisDialog.ICON_QUESTION);
                return false;
            }
            dir = selec.getPath();
            int lg = name.length();
            workDir = dir.substring(0,(dir.length()-lg));
            }
    okay = name!=null;
    if (okay) {
        this.name = name;
        this.dir = dir;
    }
    return okay;
}


void compilePlus() {
    if (!isJavac())
        return;
    if (!open("Compile Plugins..."))
        return;
    compile(dir);
}

boolean compile(String path) {
    String classpath = System.getProperty("java.class.path");
    classpath = classpath+System.getProperty("path.separator")+workDir;
```

```
File worfkfile = new File(workDir);
File[] filetab = worfkfile.listFiles();
for (int i=0; i<filetab.length; i++){
    if ((filetab[i].getName()).endsWith(".jar") ||
        (filetab[i].getName()).endsWith(".zip")){
        if (newInClasspath.equals("")){
            newInClasspath = filetab[i].getPath();
        }
        else{
            newInClasspath = newInClasspath +
            System.getProperty("path.separator") + filetab[i].getPath();
        }
    }
    //on nettoie les .class pour etre sur qu'ils soient bien tous
    //importés dans le répertoire d'hébergemenbt des plugins
    if ((filetab[i].getName()).endsWith(".class")){
        filetab[i].delete();
    }
}
output.reset();
if (!(classpath.endsWith(System.getProperty("path.separator")))){
    classpath = classpath + System.getProperty("path.separator");
}
classpath = classpath + newInClasspath;
boolean compiled = javac.compile(new String[]
    {"-deprecation","-classpath",classpath,path,"-d",pluginspath});
String s = output.toString();
boolean errors = (!compiled /*|| (s!=null && s.length()>0)*/);
if (errors){
    new Error_Editor("Compilation results",s);
    //main.update_plug_lists(name.substring(0,name.length()-6));
}
else {
    //copie du fichier .plugin (et des .jar et .zip) dans le repertoire
    //des plugins si il n'y est pas encore
    String plugname = null;
    String plugExtName = null;
    if (!(path.startsWith(pluginspath))){
        try{
            String src = (path.substring(0,(path.length()-4)))+"plugin";
            File pluginsrc = new File(src);
            FileInputStream instream = new FileInputStream(pluginsrc);
            BufferedReader reader = new BufferedReader
                    (new InputStreamReader(instream));
            plugname = reader.readLine();
            plugname = plugname.trim();
            plugExtName = pluginsrc.getName();
            String target = pluginspath+plugExtName;
            File plugintgt = new File(target);
```

```
                Utils.copyFile(pluginsrc,plugintgt);
                String readfilename = reader.readLine();
                src = workDir + (System.getProperty("file.separator").toString() );
                while (readfilename != null){
                    if (readfilename.endsWith(".jar") ||
                            readfilename.endsWith(".zip")){
                        pluginsrc = new File(src + readfilename);
                        target = pluginspath + readfilename;
                        plugintgt = new File(target);
                        Utils.copyFile(pluginsrc,plugintgt);
                        ClassPathModifier.addFile(target);
                    }
                    readfilename = reader.readLine();
                }
            }
            catch(Exception e){
                System.out.println("Exception spotted ! :");
                e.printStackTrace();
                return false;}
        }

        TelemisDialog.dialog("Compilation results",
                "Classe compilée !",
                Language.translate("OK"),
                Language.translate("Cancel"), null, TelemisDialog.ICON_QUESTION);
        myPluginExtension.update_plug_lists(plugname);
        //on les appelle avec leur vrai nom qui se trouve
        //a la premiere ligne du .plugin
        plugExtName = (plugExtName.substring(0,plugExtName.length()-7));
        myPluginExtension.addDicoEntry(plugname,plugExtName);
        myPluginExtension.update_depfile
            ((name.substring(0,name.length()-5))+".plugin");
    }
    return compiled;
  }

}
```

Classe MyPluginExtension

```
package telemis.util;

import telemis.om.image.ImageWindow;
import telemis.extention.*;

public interface MyPluginExtension extends telemis.extention.Extention {

    public void initWindow(ImageWindow imagewindow);

    public void initMenus();

    public boolean pluginExists(String plugName);

    public void update_plug_lists(String s);

    public void addDicoEntry(String realName, String fileName);

    public void update_depfile(String pluginName);

}
```

Classe MyPluginExtensionBP

```
package telemis.plugins;

import telemis.om.image.*;
import telemis.awt.*;
import telemis.util.*;

import java.awt.Font;
import java.util.*;
import java.io.*;
import javax.swing.JMenuItem;
import javax.swing.JFileChooser;

public class MyPluginExtensionBP implements MyPluginExtension {
    // INdex interface name
    String myIndex = null;

    ImageWindow myImageWindow = null;

    javax.swing.JMenu Plugins=new javax.swing.JMenu();
```

```
javax.swing.JMenuItem load_plugins= new javax.swing.JMenuItem();
javax.swing.JMenuItem delete_plugins= new javax.swing.JMenuItem();
javax.swing.JMenuItem debug_plugins= new javax.swing.JMenuItem();
javax.swing.JMenuItem compile_plugins= new javax.swing.JMenuItem();
javax.swing.JMenu installed_plugins=new javax.swing.JMenu();

String separator = (System.getProperty("file.separator").toString() );
String userdir = (System.getProperty("user.dir").toString() );
public String pluginsdir = userdir + separator + "plugins" + separator ;
Vector plugins = new Vector();
Hashtable pluginsDictionary = new Hashtable();
MyMenuActionListener myMenuActionListener = new MyMenuActionListener();
MyPluginExtensionBP me = this;

public void initWindow(ImageWindow imagewindow){
    myImageWindow = imagewindow;
}

public void initMenus(){
    Plugins.setText("Plugins");
    this.setTheMenuDisplayProperties(Plugins);
    myImageWindow.JMenuBar1.add(Plugins);
    load_plugins.setText("Load a plugin ...");
    delete_plugins.setText("Delete a plugin ...");
    compile_plugins.setText("Compile and install a plugin ...");
    debug_plugins.setText("Build a plugin's environment ...");
    installed_plugins.setText("Installed plugins ... ");
    this.setTheMenuDisplayProperties(Plugins);
    this.setTheMenuDisplayProperties(load_plugins);
    this.setTheMenuDisplayProperties(delete_plugins);
    this.setTheMenuDisplayProperties(compile_plugins);
    this.setTheMenuDisplayProperties(debug_plugins);
    this.setTheMenuDisplayProperties(installed_plugins);
    Plugins.add(load_plugins);
    Plugins.add(delete_plugins);
    Plugins.add(compile_plugins);
    Plugins.add(debug_plugins);
    Plugins.addSeparator();
    Plugins.add(installed_plugins);
    int fontSize = DefaultFont.getDefaultFontSize();
    addPluginItems ();
}

private void setTheMenuDisplayProperties(JMenuItem myMenu) {
    int fontSize = DefaultFont.getDefaultFontSize();
    myMenu.setFont(new Font("Dialog", java.awt.Font.PLAIN, fontSize));
    myMenu.addActionListener(myMenuActionListener);
}
```

```
class MyMenuActionListener implements java.awt.event.ActionListener {
    public void actionPerformed(java.awt.event.ActionEvent e) {

        if (e == null){
            System_Out_Display.addMsg("e est null");
            return;}
        Object source = e.getSource();
        if (source == null){
            System_Out_Display.addMsg("source est null");
            return;}

        if (source == compile_plugins){
            telemis.plugins.Compiler compiler = new telemis.plugins.Compiler(me, pluginsdir);
            Thread thread = new Thread(compiler);
            thread.start();
        } else
        if (source == delete_plugins){
            deletePlugin();
        } else
        if (source == load_plugins) {
            loadPlugin();
        } else
        if (source == debug_plugins) {
            telemis.plugins.Debugger debugger = new telemis.plugins.Debugger(myImageWindow);
            Thread thread = new Thread(debugger);
            thread.start();
        } else
        {
            if (installed_plugins.isMenuComponent((JMenuItem)source)){
                System_Out_Display.addMsg("plugin item");
                String classWithoutExtension = getPluginFilename(e.getActionCommand());
                System_Out_Display.addMsg(classWithoutExtension);
                runPlugin(classWithoutExtension);
                //ici il faut appeler avec le nom de la classe sans l'extension
            }
        }
    }
}


//Renvoie un tableaux avec les fichiers situés dans le répertoire des plugins
public String[] getPlugins(){
    File f = new File(pluginsdir);
    String[] tab =  f.list();
    return tab;
}
```

```
//Cette méthode construit a partir d'un tableau de strings représentant des noms
//de fichiers, un vecteur avec les noms des plugins installés ... et ajoute également
//les entrées dans le dictionnaire
public Vector fillClassVector(String[] tab){
    Vector v = new Vector();
    File f = null;
    for(int i=0; i<tab.length ; i++){
        String name = tab [i];
        String itemName = null;
        if (name.endsWith("_.plugin") && name.indexOf('.') > 0 ){
            f= new File (pluginsdir+name);
            try{
                FileInputStream inplug = new FileInputStream(f);
                BufferedReader pluginreader = new BufferedReader
                        (new InputStreamReader(inplug));
                itemName = (pluginreader.readLine()).trim();
                v.addElement(itemName);
                name = name.substring(0,(name.length()-7));
                addDicoEntry(itemName,name);
            }
            catch (Exception e){
            }
        }
        //ajout au classpath des .jar et .zip deja dans le répertoire
        //des plugins (ajouté pendant déboguage)
        try{
            if ((name.endsWith(".jar")) || (name.endsWith(".zip")) ||
                (name.endsWith(".JAR")) || (name.endsWith(".ZIP"))  ){
                ClassPathModifier.addFile(pluginsdir + name);
            }
        }
        catch (Exception e){
            e.printStackTrace();
        }
    }
    return (v);
}

//Cette méthode appelée au démarrage de l'application ajoute dans le menu
//des plugins les items des plugins installés
public void addPluginItems (){
    Vector v = fillClassVector(getPlugins());
    int lg = v.size();
    for(int i=0; i<lg ; i++){
        String s = (String)v.elementAt(i);
        JMenuItem item = new JMenuItem(s);
        this.setTheMenuDisplayProperties(item);
        installed_plugins.add(item);
        myImageWindow.addACommand(myImageWindow.REDRAW);
```

```
            plugins.addElement(s);
        }
    }

    //S'occupe de la mise à jour du menu des plugins installés après compilation
    //d'un nouveau plugin
    public void update_plug_lists(String s){
        //s = filename_to_command(s);
        int size = plugins.size();
        JMenuItem item = new JMenuItem(s);
        this.setTheMenuDisplayProperties(item);
        installed_plugins.add(item);
        myImageWindow.addACommand(myImageWindow.REDRAW);
        plugins.addElement(s);
    }

    public void remove_from_plug_list(String s){
        plugins.remove(s);
    }

    public void remove_from_menu(String s){
    //l'argument est également le vrai nom du plugin situé dans la première ligne du fichier
        int nbItem = installed_plugins.getItemCount();
        boolean goon = true;
        for (int i=0; (goon && i<nbItem) ; i++){
            if (installed_plugins.getItem(i).getActionCommand().equals(s)){
                installed_plugins.remove(i);
                goon = false;
            }
        }
    }

    //Méthode d'exécution du plugin
    public void runPlugin(String className){
        PluginClassLoader loader = new PluginClassLoader(pluginsdir);
        System_Out_Display.addMsg("loader chargé !");
        Object thePlugIn = null;
        try {
            Class laclasse = loader.loadClass(className);
            System_Out_Display.addMsg("classe créée");
            thePlugIn = laclasse.newInstance();
            System_Out_Display.addMsg("instance de plugin créée par le classloader");
            if (thePlugIn instanceof Plugin){
                System_Out_Display.addMsg("c'est bien une instance de plugin");
                ((Plugin)thePlugIn).runit(myImageWindow);
            }
            else System_Out_Display.addMsg("Stuut car le brol n'est pas une instance de Plugin");
        }
        catch (ClassNotFoundException e) {
```

```
            System.out.println("Plugin not found: "+className);
        }
        catch (InstantiationException e) {System.out.println("Unable to load plugin (ins)");}
        catch (IllegalAccessException e) {System.out.println("Unable to load plugin (acc)");}
    }

    public void deletePlugin(){
        File dirfile = new File(pluginsdir);
        JFileChooser chooser = new JFileChooser();
        chooser.setCurrentDirectory(dirfile);
        ExtensionFilter filtre;
        filtre = new ExtensionFilter();
        filtre.addExtension("_.plugin");
        filtre.setDescription(".plugin");
        chooser.setFileFilter(filtre);
        chooser.setDialogTitle("Delete a plugin");
        int res = chooser.showDialog(null,"Delete");
        if (res == JFileChooser.APPROVE_OPTION) {
            File deletefile = chooser.getSelectedFile();
            String pathin = deletefile.getPath();
            String prefix = (pathin.substring(0, (pathin.length()-7))) ;
            String fileName = deletefile.getName();
            String lu = null;
            FileInputStream instream = null;
            BufferedReader reader = null;
            try{
                instream = new FileInputStream(deletefile);
                reader = new BufferedReader(new InputStreamReader(instream));
                lu = reader.readLine();
            }
            catch (Exception e){
                return;
            }
            fileName = fileName.substring(0, (fileName.length()-7));
            //on vire de la liste des plugins
            remove_from_plug_list(lu.trim());
            //on vire du dictionnaire
            removeDicoEntry(lu.trim());
            //on vire du menu des plugins
            remove_from_menu(lu.trim());
            //on vire le .class principal
            deletefile = new File(prefix+".class");
            deletefile.delete();
            //on vire les sous-classes du .class principal
            File[] filetab = dirfile.listFiles();
            String filepath="";
            for (int i=0; i<filetab.length ; i++){
                filepath = filetab[i].getPath();
                if (filepath.startsWith(prefix+"$")){
```

```
                filetab[i].delete();
            }
        }
        //on met a jour les dépendances
        deletefile = new File(prefix+".plugin");
        Vector toClean = null;
        try{
            lu = reader.readLine();
            toClean = new Vector();
            while(lu != null){
                toClean.addElement(lu);
                lu = reader.readLine();
            }
            instream.close();
        }
        catch (Exception e){
            return;
        }

        if (toClean.size()>0){
            cleanDepFile(toClean);
        }
        //on vire le .plugin
        deletefile.delete();
        //deletage terminé !
        TelemisDialog.dialog("Plugin removal",
        "Plugin deleted",
        Language.translate("OK"),
        Language.translate("Cancel"), myImageWindow, TelemisDialog.ICON_QUESTION);
    }
}

public void cleanDepFile(Vector toClean){
    try{
        File depfile = new File(pluginsdir+"dependencies.cfg");
        Vector depcontent = new Vector();
        FileInputStream in = new FileInputStream(depfile);
        BufferedReader inreader = new BufferedReader(new InputStreamReader(in));
        String lu = inreader.readLine();
        while(lu != null){
            depcontent.addElement(lu);
            lu = inreader.readLine();
        }
        if (depcontent.size()==0){
            return;
        }
        String str=null;
        String fichier=null;
        String nbr=null;
```

```
            int intnbr;
            File temp = null;
            for (int i=0; i<toClean.size(); i++){
                str = (String)(toClean.elementAt(i));
                System.out.println(depcontent.size());
                for (int j=0; j<depcontent.size(); j++){
                    fichier = (String)(depcontent.elementAt(j));
                    j++;
                    nbr = (String)(depcontent.elementAt(j));
                    intnbr = Integer.parseInt(nbr);
                    if (str.equals(fichier)){
                        if (intnbr==1){
                        //il n'y aura plus de dépendances concernant ce fichier,
                        //il est donc a effacer
                            temp = new File(pluginsdir+fichier);
                            temp.delete();
                            depcontent.removeElementAt(j);
                            depcontent.removeElementAt(j-1);
                        }
                        else{
                            depcontent.removeElementAt(j);
                            depcontent.insertElementAt((Integer.toString(intnbr-1)),j);
                        }
                    }
                }
            }
            in.close();
            FileOutputStream out = new FileOutputStream(depfile);
            DataOutputStream outstream = new DataOutputStream(out);
            PrintWriter writer = new PrintWriter(out,true);
            String towrite=null;
            for (int z=0; z<depcontent.size(); z++){
                writer.println(((String)depcontent.elementAt(z)));
            }
            out.close();
        }
        catch (Exception e){e.printStackTrace();}

    }

    public void update_depfile(String pluginName){
        try {
            File plugin = new File(pluginsdir+pluginName);
            File depfile = new File(pluginsdir+"dependencies.cfg");
            //préparation de la lecture dans le fichier plugin
            FileInputStream inplug = new FileInputStream(plugin);
            BufferedReader pluginreader = new BufferedReader(new InputStreamReader(inplug));
            String lu = pluginreader.readLine();
            //on place le contenu de dependencies dans un vecteur
```

```
Vector depcontent = new Vector();
FileInputStream indep = new FileInputStream(depfile);
BufferedReader depreader = new BufferedReader(new InputStreamReader(indep));
String ludep = depreader.readLine();
while (ludep != null){
    depcontent.addElement(ludep);
    ludep = depreader.readLine();
}
indep.close();
//ensuite on lit le contenu du fichier plugin ligne par ligne et on met
//a jour les dépendances si nécessaire
lu = pluginreader.readLine();
int i;
boolean goon;
String fichier;
String nbr;
int intnbr;
while (lu != null){
    i = 0;
    goon = true;
    while((i<depcontent.size())&&(goon)){
        fichier = (String)depcontent.elementAt(i);
        i++;
        nbr = (String)depcontent.elementAt(i);
        intnbr = Integer.parseInt(nbr);
        if (lu.equals(fichier)){
            depcontent.removeElementAt(i);
            depcontent.insertElementAt((Integer.toString(intnbr+1)),i);
            goon = false;
        }
        i++;
    }
    if (goon){
    //aucune dépendance concernant ce fichier n'était enregistrée auparavant,
    //il faut donc l'inscrire
        depcontent.addElement(lu);
        depcontent.addElement("1");
    }
    lu = pluginreader.readLine();
}
inplug.close();
//on met enfin réellement à jour le fichier des dépendances
//depfile.delete();
//depfile.createNewFile();
FileOutputStream out = new FileOutputStream(depfile);
DataOutputStream outstream = new DataOutputStream(out);
PrintWriter writer = new PrintWriter(out,true);
String towrite=null;
for (int z=0; z<depcontent.size(); z++){
```

```
            writer.println(((String)depcontent.elementAt(z)));
        }
        out.close();

    }//fin du try
    catch (Exception e){}

}

public void loadPlugin(){
        JFileChooser chooser = new JFileChooser();
        //File sourceFile = new File("C:\\");
        //chooser.setCurrentDirectory(sourceFile);
        telemis.plugins.ExtensionFilter filtre = new telemis.plugins.ExtensionFilter();
        filtre.addExtension("_.plugin");
        filtre.setDescription("_.plugin");
        chooser.setFileFilter(filtre);
        chooser.setDialogTitle("Load a plugin");
        int res = chooser.showDialog(myImageWindow,"Load");
        if (res == JFileChooser.APPROVE_OPTION) {
            File loadfile = chooser.getSelectedFile();
            String pathin = loadfile.getPath();
            String prefix = (pathin.substring(0, (pathin.length()-7))) ;
            String fileName = loadfile.getName();
            System_Out_Display.addMsg(userdir);
            if (pluginExists(fileName)){
                TelemisDialog.dialog("Plugin already installed",
                "This plugin is already installed. Please delete it before reloading.",
                Language.translate("OK"),
                Language.translate("Cancel"), myImageWindow, TelemisDialog.ICON_QUESTION);
                return;
            }
            int lg = fileName.length();
            String workDir = pathin.substring(0,(pathin.length())-lg);
            Vector plugcontent = new Vector();
            File[] filetab = null;
            String lu = null;
            String trueName = null;
            //on vérifie que tous les fichiers nécessaires trouvent bien dans le répertoire
            try{
                FileInputStream in = new FileInputStream(loadfile);
                BufferedReader inreader = new BufferedReader(new InputStreamReader(in));
                File dirfile = new File(workDir);
                filetab = dirfile.listFiles();
                trueName = (inreader.readLine()).trim();
                lu = inreader.readLine();
                while (lu != null){
                    plugcontent.addElement(lu);
                    lu = inreader.readLine();
```

```
    }
    in.close();
}
catch (Exception e){
    return;
}
int i=0;
boolean found = false;
String lulu = null;
for (int cpt=0; cpt<plugcontent.size(); cpt++)
{
    i=0;
    lulu = (String)(plugcontent.elementAt(cpt));
    while ((i<filetab.length)&&(!found)){
        if (lulu.equals(filetab[i].getName())){//le fichier s'y trouve c ok
            found = true;
        }
        i++;
    }
    if (!found){
        TelemisDialog.dialog("Fichier manquant",
        "Un fichier est manquant à la source pour pouvoir charger ce plugin",
        Language.translate("OK"),
        Language.translate("Cancel"), myImageWindow, TelemisDialog.ICON_QUESTION);
        return;
    }
}
//si on est là tous les fichiers référencés dans le .plugin
//sont présents dans le répertoire, on peut les transférer
File inbis = null;
File out = null;
for (int z=0; z<plugcontent.size(); z++){
    String fileTransName = (String)(plugcontent.elementAt(z));
    inbis = new File(workDir+fileTransName);
    out = new File(pluginsdir+fileTransName);
    Utils.copyFile (inbis,out);
    try{
        if (fileTransName.endsWith(".jar") || fileTransName.endsWith(".zip")){
            ClassPathModifier.addFile(pluginsdir+fileTransName);
        }
    }
    catch (Exception e){
        e.printStackTrace();
    }

}
//on transfert ensuite les fichiers qui commencent par le même nom que le plugin
String pref$ = (fileName.substring(0, (fileName.length()-7)))+"$";
File plug$ = null;
```

```
            for (int a=0; a<filetab.length; a++){
                String filename = filetab[a].getName();
                if (filename.startsWith(pref$)){
                    plug$ = new File(pluginsdir+filename);
                    Utils.copyFile (filetab[a],plug$);
                }
            }
            //il reste à transférer le .class principal
            File principalin = new File(prefix+".class");
            String classfile = (fileName.substring(0, (fileName.length()-7))) ;
            classfile = classfile;
            File principalout = new File(pluginsdir+classfile+".class");
            Utils.copyFile (principalin,principalout);
            // ... et le .plugin
            File pluginout = new File(pluginsdir+fileName);
            Utils.copyFile(loadfile,pluginout);

            //reste enfin à mettre à jour l'interface graphique
            //pour que le plugin soit appelable
            update_plug_lists(trueName);
            addDicoEntry(trueName,classfile);
            update_depfile(fileName);
            TelemisDialog.dialog("Plugin loaded",
            "The plugin is loaded and ready to be used",
            Language.translate("OK"),
            Language.translate("Cancel"), myImageWindow, TelemisDialog.ICON_QUESTION);

        }
}

public boolean pluginExists(String plugName){//l'argument est un .plugin !!!
    File plugDir = new File(pluginsdir);
    if (plugDir == null){
        System_Out_Display.addMsg("plugDir est null");
    }
    File[] tabfiles = null;
    try{
        System_Out_Display.addMsg(plugDir.getPath());
        tabfiles = plugDir.listFiles();
    }
    catch (Exception e){
        System_Out_Display.addMsg(e.toString());
    }
    boolean res = false;
    int i = 0;
    while ((i<tabfiles.length)&&(!res)){
        if (plugName.equals(tabfiles[i].getName())){
            res = true;
        }
```

```
            i++;
        }
        return res;
    }

    public void addDicoEntry(String realName, String fileName){
        pluginsDictionary.put(realName,fileName);
    }

    public String getPluginFilename(String realName){
        String value = (String)(pluginsDictionary.get(realName));
        return value;
    }

    public void removeDicoEntry(String realName){
        pluginsDictionary.remove(realName);
    }

    /** extension of the Telemis interface */

    public boolean setIndex (String index) {

        myIndex = index;
        return true;
    }

    public boolean init (String args[]) {
        return true;
    }

    public boolean finish () {
        return true;
    }

    public Object execute(Object command) {
        return new java.util.Vector();
    }

    public String getIndex()
    {
        return myIndex;
    }

    public boolean loadExtention()
    {
        return true;
    }

}
```